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

(19) **World Intellectual Property Organization**
International Bureau

(43) **International Publication Date**
13 June 2024 (13.06.2024)

(10) **International Publication Number**
WO 2024/119490 A1

(51) **International Patent Classification:**
G06F 11/36 (2006.01)

(21) **International Application Number:**
PCT/CN2022/137991

(22) **International Filing Date:**
09 December 2022 (09.12.2022)

(25) **Filing Language:** English

(26) **Publication Language:** English

(71) **Applicant: EBAY INC.** [US/US]; 2025 Hamilton Avenue, San Jose, California 95125 (US).

(72) **Inventor; and**
(71) **Applicant** *(for SC only)*: **CHEN, Wei** [CN/CN]; 2nd Floor, Tower 3, 88 Keyuan Road, Pudong, Shanghai 201203 (CN).

(74) **Agent: CHINA SCIENCE PATENT & TRADEMARK AGENT LTD.**; Suite 4-312, No. 87, West 3rd Ring North Rd., Haidian District, Beijing 100089 (CN).

(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, CV, CZ, DE, DJ, DK, DM, DO, DZ, EC, EE, EG, ES, FI, GB, GD, GE, GH, GM, GT, HN, HR, HU, ID, IL, IN, IQ, IR, IS, IT, JM, JO, JP, KE, KG, KH, KN, KP, KR, KW, KZ, LA, LC, LK, LR, LS, LU, LY, MA, MD, 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, ST, SV, SY, TH, TJ, TM, TN, TR, TT, TZ, UA, UG, US, UZ, VC, VN, WS, ZA, ZM, ZW.

(84) **Designated States** *(unless otherwise indicated, for every kind of regional protection available)*: ARIPO (BW, CV, GH, GM, KE, LR, LS, MW, MZ, NA, RW, SC, 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, ME, 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))*

(54) **Title:** FAULT INJECTION IN A NOTIFICATION PLATFORM

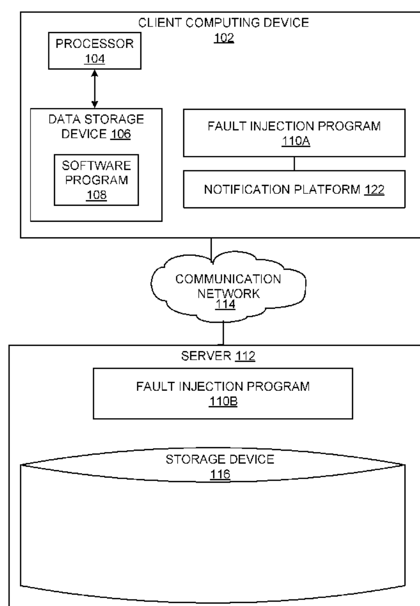


FIG. 1

(57) **Abstract:** The present invention may include an embodiment that determines a code of an application. The embodiment may receive one or more parameters from a GUI. The embodiment may convert the one or more parameters to an instrumentation code in the code of the application, where the instrumentation code simulates one or more faults in a system and comprises annotation. The embodiment may execute the instrumentation code in the code of the application to identify resilience of the application.

WO 2024/119490 A1

FAULT INJECTION IN A NOTIFICATION PLATFORM

BACKGROUND

[0001] The present invention relates, generally, to the field of computing, and more particularly to building resilient computer systems.

[0002] Fault injection is the process by which a user deliberately introduces faults into the system. Thus, the user may observe the system behavior with the injected faults to identify the weakness of the system. Within the industry, fault injection is a common practice as a means to build fault-tolerant, resilient systems.

SUMMARY

[0003] According to one embodiment, a method, computer system, and computer program product for injecting faults in the application level is provided. The present invention may include an embodiment that determines a code of an application. The embodiment may receive one or more parameters from a GUI. The embodiment may convert the one or more parameters to an instrumentation code in the code of the application, where the instrumentation code simulates one or more faults in a system and comprises annotation. The embodiment may execute the instrumentation code in the code of the application to identify resilience of the application.

BRIEF DESCRIPTION OF THE DRAWINGS

[0004] The present technology is described in detail below with reference to the attached drawing figures, wherein:

[0005] FIG. 1 is a block diagram depicting an exemplary networked computer environment according to one of the embodiments;

[0006] FIG. 2 is an embodiment depicting communications between notification service and faults injection agent according to one of the embodiments;

[0007] FIG. 3 depicts a block or interrupt instrumentation according to one of the embodiments;

[0008] FIG. 4 depicts change of method parameters instrumentation according to one of the embodiments;

[0009] FIG. 5 depicts value replacement instrumentation according to one of the embodiments;

[0010] FIG. 6 depicts an example of instrumentation using a class loader that instruments and annotates the code;

[0011] FIG. 7 depicts an instrumentation logic with a string literal according to one of the embodiments;

[0012] FIG. 8 depicts a timeout exception instrumentation according to one of the embodiments;

[0013] FIG. 9 depicts a customized class loader diagram according to an example embodiment; and

[0014] FIG. 10 depicts Graphical User Interface (GUI) of the fault injection according to one of the embodiments.

DETAILED DESCRIPTION

[0015] Detailed embodiments of the claimed structures and methods are disclosed herein; however, it can be understood that the disclosed embodiments are merely illustrative of the claimed structures and methods that may be embodied in various forms. This invention may, however, be embodied in many different forms and should not be construed as limited to the exemplary embodiments set forth herein. In the description, details of well-known features and techniques may be omitted to avoid unnecessarily obscuring the presented embodiments.

[0016] The present invention may be a system, a method, and/or a computer program product at any possible technical detail level of integration. The computer program product may include a computer readable storage medium (or media) having computer readable program instructions thereon for causing a processor to carry out aspects of the present invention.

[0017] The computer readable storage medium can be a tangible device that can retain and store instructions for use by an instruction execution device. The computer readable storage medium may be, for example, but is not limited to, an electronic storage device, a magnetic storage device, an optical storage device, an electromagnetic storage device, a semiconductor storage device, or any suitable combination of the foregoing. A non-exhaustive list of more specific examples of the computer readable storage medium includes the following: a portable computer diskette, a hard disk, a random access memory (RAM), a read-only memory (ROM), an erasable programmable read-only memory (EPROM or Flash memory), a static random access memory (SRAM), a portable compact disc read-only memory (CD-ROM), a digital versatile disk (DVD), a memory stick, a floppy disk, a mechanically encoded device such as

punch-cards or raised structures in a groove having instructions recorded thereon, and any suitable combination of the foregoing. A computer readable storage medium, as used herein, is not to be construed as being transitory signals *per se*, such as radio waves or other freely propagating electromagnetic waves, electromagnetic waves propagating through a waveguide or other transmission media (e.g., light pulses passing through a fiber-optic cable), or electrical signals transmitted through a wire.

[0018] Computer readable program instructions described herein can be downloaded to respective computing/processing devices from a computer readable storage medium or to an external computer or external storage device via a network, for example, the Internet, a local area network, a wide area network and/or a wireless network. The network may comprise copper transmission cables, optical transmission fibers, wireless transmission, routers, firewalls, switches, gateway computers and/or edge servers. A network adapter card or network interface in each computing/processing device receives computer readable program instructions from the network and forwards the computer readable program instructions for storage in a computer readable storage medium within the respective computing/processing device.

[0019] Computer readable program instructions for carrying out operations of the present invention may be assembler instructions, instruction-set-architecture (ISA) instructions, machine instructions, machine dependent instructions, microcode, firmware instructions, state-setting data, configuration data for integrated circuitry, or either source code or object code written in any combination of one or more programming languages, including an object oriented programming language such as Smalltalk, C++, or the like, and procedural programming languages, such as

the "C" programming language or similar programming languages. The computer readable program instructions may execute entirely on the user's computer, partly on the user's computer, as a stand-alone software package, partly on the user's computer and partly on a remote computer or entirely on the remote computer or server. In the latter scenario, the remote computer may be connected to the user's computer through any type of network, including a local area network (LAN) or a wide area network (WAN), or the connection may be made to an external computer (for example, through the Internet using an Internet Service Provider). In some embodiments, electronic circuitry including, for example, programmable logic circuitry, field-programmable gate arrays (FPGA), or programmable logic arrays (PLA) may execute the computer readable program instructions by utilizing state information of the computer readable program instructions to personalize the electronic circuitry, in order to perform aspects of the present invention.

[0020] Aspects of the present invention are described herein with reference to flowchart illustrations and/or block diagrams of methods, apparatus (systems), and computer program products according to embodiments of the invention. It will be understood that each block of the flowchart illustrations and/or block diagrams, and combinations of blocks in the flowchart illustrations and/or block diagrams, can be implemented by computer readable program instructions.

[0021] These computer readable program instructions may be provided to a processor of a general purpose computer, special purpose computer, or other programmable data processing apparatus to produce a machine, such that the instructions, which execute via the processor of the computer or other programmable data processing apparatus, create means for implementing the

functions/acts specified in the flowchart and/or block diagram block or blocks. These computer readable program instructions may also be stored in a computer readable storage medium that can direct a computer, a programmable data processing apparatus, and/or other devices to function in a particular manner, such that the computer readable storage medium having instructions stored therein comprises an article of manufacture including instructions which implement aspects of the function/act specified in the flowchart and/or block diagram block or blocks.

[0022] The computer readable program instructions may also be loaded onto a computer, other programmable data processing apparatus, or other device to cause a series of operational steps to be performed on the computer, other programmable apparatus or other device to produce a computer implemented process, such that the instructions which execute on the computer, other programmable apparatus, or other device implement the functions/acts specified in the flowchart and/or block diagram block or blocks.

[0023] The flowchart and block diagrams in the Figures illustrate the architecture, functionality, and operation of possible implementations of systems, methods, and computer program products according to various embodiments of the present invention. In this regard, each block in the flowchart or block diagrams may represent a module, segment, or portion of instructions, which comprises one or more executable instructions for implementing the specified logical function(s). In some alternative implementations, the functions noted in the blocks may occur out of the order noted in the Figures. For example, two blocks shown in succession may, in fact, be executed substantially concurrently, or the blocks may sometimes be executed in the

reverse order, depending upon the functionality involved. It will also be noted that each block of the block diagrams and/or flowchart illustration, and combinations of blocks in the block diagrams and/or flowchart illustration, can be implemented by special purpose hardware-based systems that perform the specified functions or acts or carry out combinations of special purpose hardware and computer instructions.

[0024] The eBay notification platform team may utilize an idea of fault injection from a different perspective in eBay’s notification platform. This platform is responsible for pushing platform notifications to third party applications to provide the latest changes in item price, item stock status, payment status and more. It is a highly distributed and large-scale system relying on many external dependencies, including distributed store, message queue, push notification endpoints and others. Any faults in these dependent services will directly impact the stability of the system, so the notification platform may be utilized to run experiments in the system containing the failures of these dependencies, in order to understand the behaviors and mitigate any weaknesses in the application and the corresponding system that executes the application.

[0025] To achieve system and application resilience, the faults may be injected by simulation in the application level by the code instrumentation as described below. This is a novel approach in the industry that enables experimentation on the mission-critical system with different kinds of deliberately designed faults to improve system stability and overall resilience to errors.

[0026] There is no unified way to inject these faults into an application. The most straightforward and ubiquitous way to do so in the industry is to create real faults directly in the underlining system. For example, to introduce the *http* disconnection or timeout error, one option is to turn off the network or shut down the downstream services temporarily; to introduce the disk full error, one option is to create a bunch of files in the file system. Typical user may think of this as fault injection at the infrastructure level, and if the user thinks about it from this perspective, it will inevitably drive the user to create the concrete faults for the infrastructure resources. Creating faults may directly harm the infrastructure resources. For example, turning off the network obviously causes multiple issues, as mentioned in the previous example. When the resource is shared, this haring may introduce extra impacts and risks to other services depending on that resource. If the resource is dedicated, it may increase the cost.

[0027] Alternatively, a different way to approach this problem may be utilized. Instead of creating faults at the infrastructure level, what if we may generate the faults at the application level? This would allow an user to simulate the faults he would like to use with the application API leveraged to communicate with the infrastructure resources. For example, to inject the *http* timeout fault, a latency in the http client library may be added; to simulate the internal service error, the response code with 500 *http* status code may be simulated. The faults may be restricted to the API level and would not harm the underlying infrastructure resources. In this way, we may create an affordable, secure and reusable software-based solution to do fault injection.

[0028] According to one embodiment, a Java-based application such as Java agent may be provided. Within the agent, instrumented the class files of the client libraries for the

dependent services to introduce different kinds of faults may be instrumented and defined. Furthermore, the introduced faults may be raised when our service communicates with the underlying resource through the instrumented API. The faults do not really happen in our dependent services, owing to the changed codes, but the effect is simulated, enabling experimentation without risk to the system.

[0029] Referring to Fig. 1, an exemplary networked computer environment 100 is depicted, according to at least one embodiment. The networked computer environment 100 may include client computing device 102 and a server 112 interconnected via a communication network 114. According to at least one implementation, the networked computer environment 100 may include a plurality of client computing devices 102 and servers 112, of which only one of each is shown for illustrative brevity.

[0030] The communication network 114 may include various types of communication networks, such as a wide area network (WAN), local area network (LAN), a telecommunication network, a wireless network, a public switched network and/or a satellite network. The communication network 114 may include connections, such as wire, wireless communication links, or fiber optic cables. It may be appreciated that Fig. 1 provides only an illustration of one implementation and does not imply any limitations with regard to the environments in which different embodiments may be implemented. Many modifications to the depicted environments may be made based on design and implementation requirements.

[0031] Client computing device 102 may include a processor 104 and a data storage device 106 that is enabled to host and run a software program 108, notification platform 122 and a fault injection program 110A and communicate with the server 112 via the communication network 114, in accordance with one embodiment of the invention. Client computing device 102 may be, for example, a mobile device, a telephone, a personal digital assistant, a netbook, a laptop computer, a tablet computer, a desktop computer, or any type of computing device capable of running a program and accessing a network.

[0032] The server computer 112 may be a laptop computer, netbook computer, personal computer (PC), a desktop computer, or any programmable electronic device or any network of programmable electronic devices capable of hosting and running a fault injection program 110B and a database 116 and communicating with the client computing device 102 via the communication network 114, in accordance with embodiments of the invention. The server 112 may also operate in a cloud computing service model, such as Software as a Service (SaaS), Platform as a Service (PaaS), or Infrastructure as a Service (IaaS). The server 112 may also be located in a cloud computing deployment model, such as a private cloud, community cloud, public cloud, or hybrid cloud.

[0033] According to the present embodiment, the fault injection program 110A, 110B may be a program capable of instrumentation and annotation of a code of a software program 108 to inject faults and analyze the effect of the faults on the software program 108.

[0034] Referring now to Fig. 2, communications between notification service and faults injection agent according to one of the embodiments are depicted. To simulate the faults for the client libraries by instrumentation is challenging. Our main task is to force the invoked methods to experience failures. One method to do this is to inject failure directly into the method, by, for example, throwing the exception in the method body. The other method calls for changing the value or the state of the input parameters to drive the method to go to the failure execution path. There are three instrumentation patterns in our project.

[0035] Referring now to Fig. 3, a block or interrupt instrumentation according to one of the embodiments is depicted. This type of instrumentation is straightforward in that the application programming interface (API) may be utilized to throw exceptions or sleep for a specific period of time to simulate the desired error or timeout.

[0036] Referring now to Fig.4 a change of method parameters instrumentation according to one of the embodiments is depicted. Under some circumstances, the simulation of faults may depend on the specific state of the input parameters. For example, the below method logic is depending on the return value of the *response.getStatusCode()*. If the value does not equal 200, the failure logic may be triggered. So if a user wants to simulate the faults with failure code, then we need to find a way to change the state of response which will be returned from the *response.getStatusCode().*

[0037] The way we achieved this is to add the instrumented code snippet to throw a specific defined exception and let the exception carry the response code we need to simulate. Meanwhile, we instrument the method by adding the try-catch block to the method to specifically catch the exception we throw and return the code in the catch block. By doing so, we've changed the method execution path to return the designated value.

[0038] Referring now to Fig. 5 a value replacement instrumentation according to one of the embodiments is depicted. In contrast to the example in Fig. 4, sometimes the method logic will depend on the value of the parameters. So if you want to simulate a fault, then you need to change the value of the input parameter. To change the value of the parameter, we need to know the name of the parameter first and inject the code to replace the value for the parameter with its name. This is not easy because the parameter name can only be known in the runtime. So we leverage the Java reflection to get the names of the parameters in the runtime.

[0039] Referring now to Fig. 6 an example of instrumentation using a class loader that instruments and annotates the code is depicted. To implement the above three types of instrumentations, a Java agent may be created. According to one of the embodiments, the agent, may implement and utilize a classloader which instruments the code of the methods that are leveraged in the application code. In another embodiments, an annotation may be added to indicate which method is instrumented and put the instrumentation logic in the methods that is annotated as depicted in the Fig. 6.

[0040] Referring now to Fig. 7 depicts an instrumentation logic with a string literal according to one of the embodiments is depicted. In the depicted code snippet, we'd like to provide the instrumentation logic for *org.asynchttpclient.providers.netty.future.NettyResponseFuture.done()*. According to one of the embodiments, a new method with the same signature may be created, that may be annotated by *@Enforce* which may be the user-defined Java annotation that may be used to indicate the instrumentation logic for fault injection. The annotation may have two fields: value and type. The value field is the class name of the method we want to instrument. (the type field is discussed below). When the agent is loaded, the defined class loader may find all the methods

annotated by *@Enforce* and inject the instrumentation logic defined in the methods to the methods to be instrumented. The type field of *@Enforce* may have the two values runtime (default value) and static. In the above example, the instrumentation logic may be implemented with the Java code. But there may be a need to provide the instrumentation logic with a string literal as depicted in the Fig. 7.

[0041] Referring now to Fig. 8 depicts a timeout exception instrumentation according to one of the embodiments is depicted. According to one of the embodiments, we may instrument the timeout exception of *org.asynchttpclient.providers.netty.future.NettyResponseFuture.awaitUpdate(final int lastVersion, final long timeoutMs)*. In the *awaitUpdate*, it leverages the *timeoutMs* to calculate the d*eadlineMs*. So if we can change the value of the passed t*imeoutMs*, we can simulate the timeout exception for this method. As we discussed before, we will define a new method with the same name of *awaitUpdate* and annotated by *@Enforce with value = org.asynchttpclient.providers.netty.future.NettyResponseFuture and type = 'stat*ic'. When we specify the type = 'static', it means the instrumentation logic will be in the form of a string literal instead of the Java code, as in the previous example. The reason to do so is that we are only able to determine the name of method parameters in the runtime. To be more specific, we only know the name of the second parameter of *awaitUpdate* in the runtime. What we do is to identify the parameter names with Java reflection during class loading and pass it into the corresponding method which provides instrumentation logic. So in the Fig. 8 example, the *params = 'lastVersion,timeoutMs'*.

[0042] Referring now to Fig. 9 depicts a customized class loader diagram according to an example embodiment is depicted. According to an example embodiment, the instrumentation

may be implemented using instrumentation logic in the Java agent. However, we still need to create a customized class loader to inject the instrumentation logic in the target methods of the client libraries into which we want to inject faults. The class loader may leverages Javassist or similar technology , and utilize the instrument library, which can manipulate the Java or other programming language bytecode to transform the class files of the target methods to include the defined faults.

[0043] With the above implementation described, we have injected the faults by instrumentation for the below client libraries of the three resources we are depending on.

- Push Notification Endpoints:
  - Client lib: async-http-client 1.8.3
  - Fault types:
    - Timeout
    - Exception
    - Response status code
- Message Queue:
  - Client lib: kafka-client 2.5.1
  - Fault types:
    - Timeout
    - Exception
- Distributed Store (built in-house by eBay):
  - Client lib: monster-java-client 3.4.4.2-RELEASE
  - Fault types:
    - Timeout
    - Exception

[0044] Referring now to Fig. 10 a Graphical User Interface (GUI) of the fault injection according to one of the embodiments is depicted. To dynamically change the configuration for the fault injections in the runtime, we have implemented a configure management console in the Java agent. As our service is a web application, we can instrument the javax.servlet.http.HttpServlet.service(HttpServletRequest, HttpServletResponse) to expose the

endpoints for the configure management. The endpoint will render a configuration page to let developers configure the attributes of the fault injection in the runtime. For example, a developer could globally enable or disable the fault injection and other subtypes of the faults; for example, a timeout for AyncHttpClient.

[0045] We will expand the scope of the application-level fault injection in more client libraries and fault categories to diversify the scenarios of experiments for our services under different kinds of fault circumstances in the future applications. Meanwhile, as the configuration of the faults setting through the configuration management console can only be triggered at the instance level, we will find a way to broadcast the changes across the cluster.

APPENDIX

Background

It might sound paradoxical to deliberately break something we're trying to fix, but sometimes, that's the most efficient method to do it. Fault injection is the process by which we deliberately introduce faults into the system. We can observe the system behavior with the injected faults to identify the weakness of the system. Within the industry, fault injection is a common practice as a means to build fault-tolerant, resilient systems.

The eBay notification platform team practiced the idea of fault injection from a different perspective in eBay's notification platform. This platform is responsible for pushing platform notifications to third party applications to provide the latest changes in item price, item stock status, payment status and more. It is a highly distributed and large-scale system relying on many external dependencies, including distributed store, message queue, push notification endpoints and others. Any faults in these dependent services will directly impact the stability of the system, so it's quite valuable to run experiments in the system containing the failures of these dependencies, in order to understand the behaviors and mitigate any weaknesses.

To achieve this, the faults are injected by simulation in the application level by the code instrumentation. As far as we know, we are the very first in the industry who practice this idea officially and widely to experiment on the mission-critical system with different kinds of deliberately designed faults.

Application Level *vs* Infrastructure Level

There is no unified way to inject these faults. The most straightforward and ubiquitous way to do so in the industry is to create real faults directly. For example, to introduce the *http* disconnection or timeout error, one option is to turn off the network or shut down the downstream services temporarily; to introduce the disk full error, one option is to create a bunch of files in the file system. We can think of this as fault injection at the infrastructure level, and if we think about it from this perspective, it will inevitably drive us to create the concrete faults for the infrastructure resources. Creating faults will directly harm the infrastructure resources; turning off the network obviously causes many issues, in the previous example. If the resource is shared, this will introduce extra impacts and risks to other services depending on it. If the resource is dedicated, it will increase the cost.

But there is a different way to approach this problem: Instead of creating faults at the infrastructure level, what if we created them at the application level? This would allow us to simulate the faults we'd like to use with the application API leveraged to talk with the infrastructure resources. For example, to inject the *http* timeout fault, we add the latency in the http client library; to simulate the internal service error, we simulate the response code with 500 *http* status code. The faults are restricted to the API level and do no harm to the underlying infrastructure resources. In this way, we've found an affordable, secure and reusable way to do fault injection.

As our service is a Java-based application, we've provided a Java agent. Within it, we instrumented the class files of the client libraries for the dependent services to introduce different kinds of faults we defined. The introduced faults are raised when our service communicates with the underlying resource through the instrumented API. The faults do not really happen in our dependent services, owing to the changed codes, but the effect is simulated, enabling us to experiment without risk.

(See Fig. 2)

Instrumentation

- To simulate the faults for the client libraries by instrumentation is challenging. Our main task is to force the invoked methods to experience failures. One method to do this is to inject failure directly into the method, by, for example, throwing the exception in the method body. The other method calls for changing the value or the state of the input parameters to drive the method to go to the failure execution path. There are three instrumentation patterns in our project.

1. Block or Interrupt the method logic

This type of instrumentation is straightforward in that the API can throw exceptions or sleep for a specific period of time to simulate the error or timeout.
(See Fig. 3)

2. Change the state of method parameters

Under some circumstances, the simulation of faults will depend on the specific state of the input parameters. For example, the below method logic is depending on the return value of the *response.getStatusCode()*. If the value does not equal 200, the failure logic will be triggered. So if we want to simulate the faults with failure code, then we need to find a way to change the state of response which will be returned from the *response.getStatusCode()*.

The way we achieved this is to add the instrumented code snippet to throw a specific defined exception and let the exception carry the response code we need to simulate. Meanwhile, we instrument the method by adding the try-catch block to the method to specifically catch the exception we throw and return the code in the catch block. By doing so, we've changed the method execution path to return the designated value.
(See Fig. 4)

3. Replace the value of method parameters

In contrast to the above example, sometimes the method logic will depend on the value of the parameters. So if you want to simulate a fault, then you need to change the value of the input parameter. To change the value of the parameter, we need to know the name of the parameter first and inject the code to replace the value for the parameter with its name. This is not easy because the parameter name can only be known in the runtime. So we leverage the Java reflection to get the names of the parameters in the runtime.
(See Fig. 5)

To implement the above three types of instrumentation, we have created a Java agent. In the agent, we have implemented a classloader which will instrument the code of the methods leveraged in the application code. We also created an annotation to indicate which method will be instrumented and put the instrumentation logic in the methods annotated. Here's an example:
(See Fig. 6)

In the above code snippet, we'd like to provide the instrumentation logic for *org.asynchttpclient.providers.netty.future.NettyResponseFuture.done()*. So what we do is to create a new method with the same signature, and make it be annotated by @Enforce which is the user-defined Java annotation used to indicate the instrumentation logic for fault injection.

The annotation has two fields: value and type. The value field is the class name of the method we want to instrument. (We'll discuss the type field shortly.) When the agent is loaded, the defined class loader will find all the methods annotated by @Enforce and inject the instrumentation logic defined in the methods to the methods to be instrumented. The type field of @Enforce has the two values runtime(default value) and static. In the above example, we implement the instrumentation logic with the Java code. But there remains a chance that we might need to provide the instrumentation logic with a string literal. Here's an example of that: (See Fig. 7)

Let's say we want to instrument the timeout exception of *org.asynchttpclient.providers.netty.future.NettyResponseFuture.awaitUpdate(final int lastVersion, final long timeoutMs)*. In the awaitUpdate, it leverages the timeoutMs to calculate the deadlineMs. So if we can change the value of the passed timeoutMs, we can simulate the timeout exception for this method. As we discussed before, we will define a new method with the same name of awaitUpdate and annotated by @Enforce with value = *org.asynchttpclient.providers.netty.future.NettyResponseFuture* and type = 'static'. When we specify the type = 'static', it means the instrumentation logic will be in the form of a string literal instead of the Java code, as in the previous example. The reason to do so is that we are only able to determine the name of method parameters in the runtime. To be more specific, we only know the name of the second parameter of awaitUpdate in the runtime. What we do is to identify the parameter names with Java reflection during class loading and pass it into the corresponding method which provides instrumentation logic. So in the below example, the params = 'lastVersion,timeoutMs'.
(See Fig. 8)

Customized Class Loader

Now we have implemented the instrumentation logic in the Java agent. However, we still need to create a customized class loader to inject the instrumentation logic in the target methods of the client libraries into which we want to inject faults. The class loader leverages Javassist, the instrument library, which can manipulate the Java bytecode to transform the class files of the target methods to include the defined faults.
(See Fig. 9)

With the above implementation described, we have injected the faults by instrumentation for the below client libraries of the three resources we are depending on.

- Push Notification Endpoints:
    - Client lib: async-http-client 1.8.3
    - Fault types:
        - Timeout
        - Exception

        - Response status code
- Message Queue:
    - Client lib: kafka-client 2.5.1
    - Fault types:
        - Timeout
        - Exception
- Distributed Store (built in-house by eBay):
    - Client lib: monster-java-client 3.4.4.2-RELEASE
    - Fault types:
        - Timeout
        - Exception

Configuration Management

To dynamically change the configuration for the fault injections in the runtime, we have implemented a configure management console in the Java agent. As our service is a web application, we can instrument the *javax.servlet.http.HttpServlet.service(HttpServletRequest, HttpServletResponse)* to expose the endpoints for the configure management. The endpoint will render a configuration page to let developers configure the attributes of the fault injection in the runtime. For example, a developer could globally enable or disable the fault injection and other subtypes of the faults; for example, a timeout for *AyncHttpClient*.
(See Fig. 10)

What's Next

We will expand the scope of the application level fault injection in more client libraries and fault categories to diversify the scenarios of experiments for our services under different kinds of fault circumstances. Meanwhile, as the configuration of the faults setting through the configuration management console can only be triggered at the instance level, we will find a way to broadcast the changes across the cluster.

CLAIMS

What is claimed is:

1. A processor-implemented method for fault injection, the method comprising:

determining a code of an application;

receiving one or more parameters from a GUI;

converting the one or more parameters to an instrumentation code in the code of the application, wherein the instrumentation code simulates one or more faults in a system and comprises annotation; and

executing the instrumentation code in the code of the application to identify resilience of the application.

2. The method of claim 1, wherein the instrumentation code comprises a specific period of time to simulate an error or timeout.

3. The method of claim 1, wherein the instrumentation code comprises a snippet with a defined exception to carry a response code.

4. The method of claim 1, wherein the instrumentation code comprises a code to replace a value of a parameter during runtime, wherein the parameter is identified using an agent.

5. A computer system for fault injection, the computer system comprising:

one or more processors, one or more computer-readable memories, one or more computer-readable tangible storage medium, and program instructions stored on at least one of

the one or more tangible storage medium for execution by at least one of the one or more processors via at least one of the one or more memories, wherein the computer system is capable of performing a method comprising:

determining a code of an application;

receiving one or more parameters from a GUI;

converting the one or more parameters to an instrumentation code in the code of the application, wherein the instrumentation code simulates one or more faults in a system and comprises annotation; and

executing the instrumentation code in the code of the application to identify resilience of the application.

6. The method of claim 5, wherein the instrumentation code comprises a specific period of time to simulate an error or timeout.

7. The method of claim 5, wherein the instrumentation code comprises a snippet with a defined exception to carry a response code.

8. The method of claim 5, wherein the instrumentation code comprises a code to replace a value of a parameter during runtime, wherein the parameter is identified using an agent.

9. A computer program product for fault injection, the computer program product comprising:

one or more computer-readable tangible storage medium and program instructions stored on at least one of the one or more tangible storage medium, the program instructions executable by a processor, the program instructions comprising:

program instructions to determine a code of an application;

program instructions to receive one or more parameters from a GUI;

program instructions to convert the one or more parameters to an instrumentation code in the code of the application, wherein the instrumentation code simulates one or more faults in a system and comprises annotation; and

program instructions to execute the instrumentation code in the code of the application to identify resilience of the application.

10. The computer program product of claim 9, wherein the instrumentation code comprises a specific period of time to simulate an error or timeout.

11. The computer program product of claim 9, wherein the instrumentation code comprises a snippet with a defined exception to carry a response code.

12. The computer program product of claim 9, wherein the instrumentation code comprises a code to replace a value of a parameter during runtime, wherein the parameter is identified using an agent.

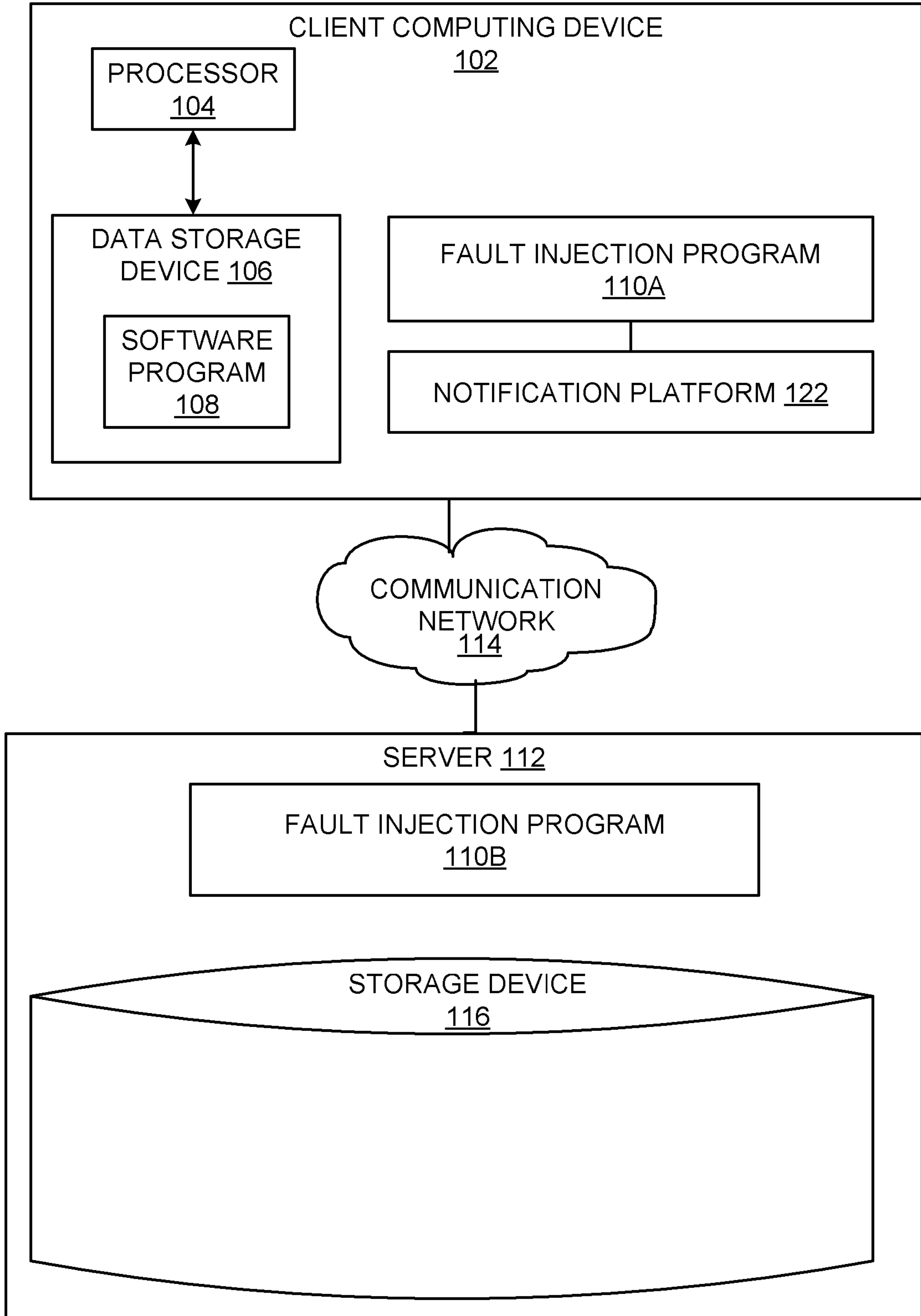


FIG. 1

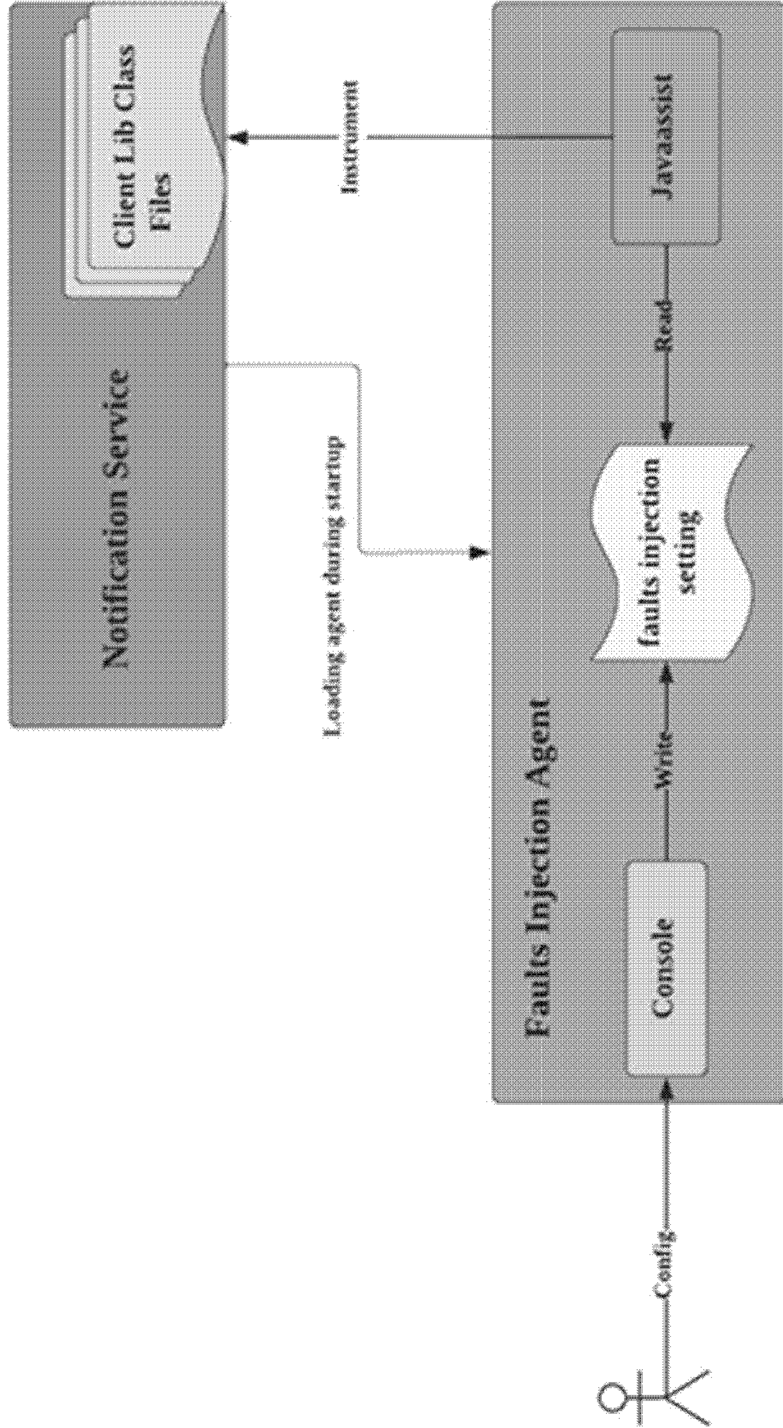


FIG. 2

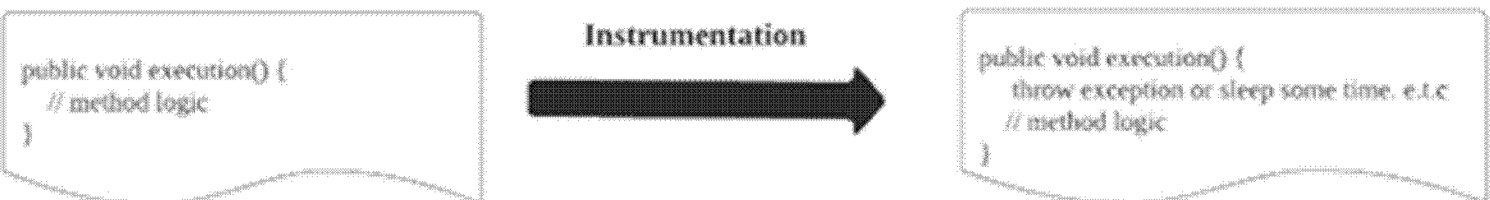


FIG. 3

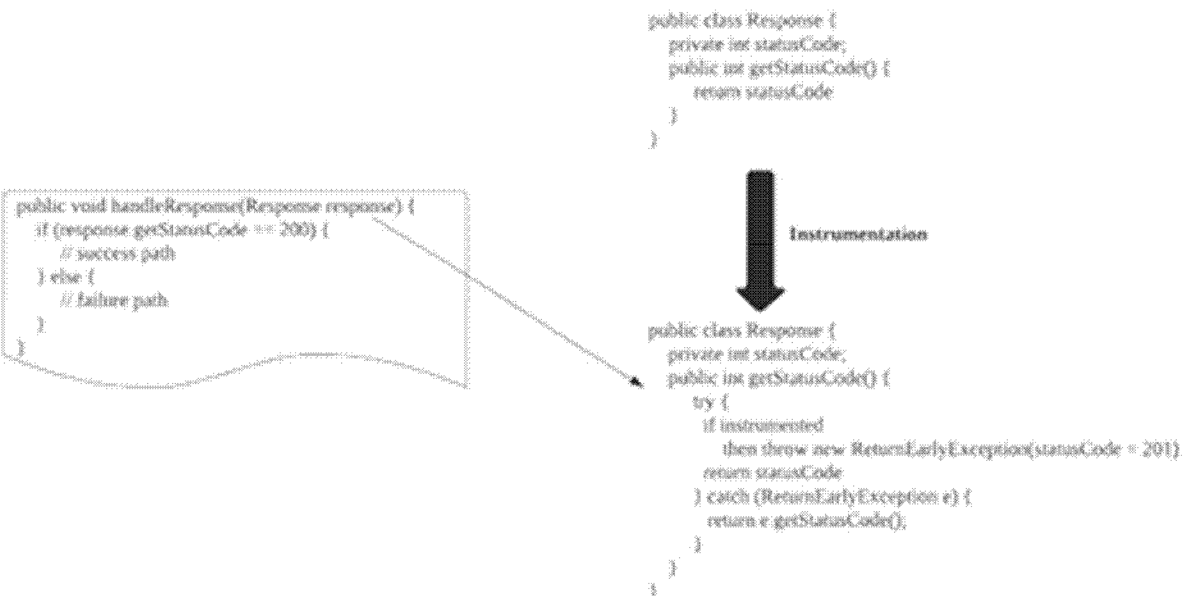
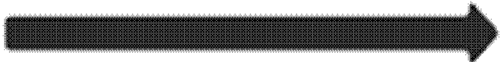


FIG. 4

```
public void handleValue(int value) {
    if (value == 0) {
        // success path
    } else {
        // failure path
    }
}
```




```
public void handleValue(int value) {
    value = 1;
    if (value == 0) {
        // success path
    } else {
        // failure path
    }
}
```

FIG. 5

```
@Enforce(value = "org.asynchttpclient.providers.netty.future.NettyResponseFuture")
public static void done(Object[] args) {
    long latency = GlobalState.getInstance().getAeroTimeout();
    if (latency > 0) {
        try {
            Thread.sleep(latency);
        } catch (Exception e) { }
    }
}
```

FIG. 6

```
public synchronized void awaitUpdate(final int lastVersion, final long timeoutMs) throws InterruptedException {
    long currentTimeMs = time.milliseconds();
    long deadlineMs = currentTimeMs + timeoutMs < 0 ? Long.MAX_VALUE : currentTimeMs + timeoutMs;
    time.waitObject(this, () -> {
        // Throw fatal exceptions, if there are any. Recoverable topic errors will be handled by the caller.
        maybeThrowFatalException();
        return updateVersion() > lastVersion || isClosed();
    }, deadlineMs);
    if (isClosed())
        throw new KafkaException("Requested metadata update after close");
}
```

FIG. 7

```
@Enforce(value = "org.apache.kafka.clients.producer.internals.ProducerMetadata", type = "static")
public static String awaitUpdate(String params) {
    String v = params.split(",")[1];
    String invocation = "\nlong timeout = com.ebay.esp.faultinject.manage.states.GlobalState.getInstance().getKafkaProducerTimeout();\n" +
            "\nSystem.out.println(timeout);\n" +
            "if (timeout > 0) {\n" +
            v + " = timeout;\n}";
    return invocation;
}
```

FIG. 8

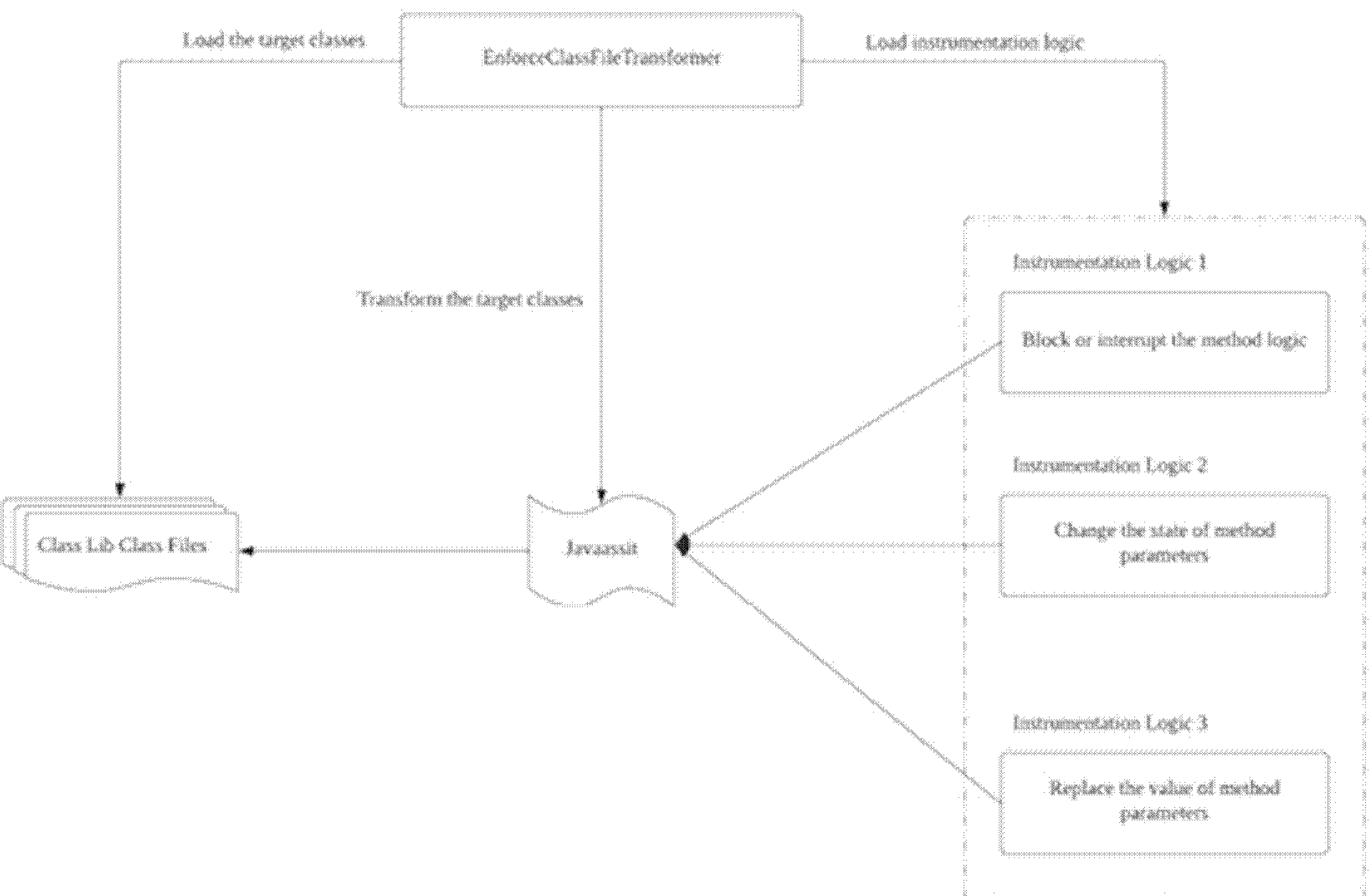


FIG. 9

Global setting:

☐ Disable All

Sample Rate 100

AsyncHttpClient:

☐ Exception for request

☐ Request timeout:

☐ Http Status Code:

Monstor Client:

☐ Exception for request

☐ Request timeout:

Kafka Client:

☑ Producer sending timeout: 12

Submit

FIG. 10

INTERNATIONAL SEARCH REPORT

International application No.

PCT/CN2022/137991

A. CLASSIFICATION OF SUBJECT MATTER

G06F11/36(2006.01)i

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)

IPC: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)

DWPI,CNKI,CNTXT,CNABS,ENTXT,ENTXTC: application, code, parameter, instrumentation, fault, resilience

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
X	CN 104834590 A (TENCENT TECHNOLOGY SHENZHEN CO., LTD.) 12 August 2015 (2015-08-12) paragraphs 0023-0061 in the description	1-12
A	CN 105095087 A (WUHAN QIMING LIANCHUANG INFORMATION TECHNOLOGY CO., LTD.) 25 November 2015 (2015-11-25) the whole document	1-12
A	US 2008178044 A1 (SHOWALTER JAMES L ET AL.) 24 July 2008 (2008-07-24) the whole document	1-12

☐ 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
- "D" document cited by the applicant in the international application
- "E" earlier application or patent but published on or after the international filing date
- "L" document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)
- "O" document referring to an oral disclosure, use, exhibition or other means
- "P" document published prior to the international filing date but later than the priority date claimed
- "T" later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention
- "X" document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone
- "Y" document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art
- "&" document member of the same patent family

Date of the actual completion of the international search	Date of mailing of the international search report
22 August 2023(22.08.2023)	**23 August 2023**

Name and mailing address of the ISA/CN

CHINA NATIONAL INTELLECTUAL PROPERTY ADMINISTRATION
6, Xitucheng Rd., Jimen Bridge, Haidian District, Beijing 100088, China

Authorized officer

CUI,LiYan

Telephone No. **(+86) 62411682**

INTERNATIONAL SEARCH REPORT
Information on patent family members

International application No.
PCT/CN2022/137991

Patent document cited in search report			Publication date (day/month/year)	Patent family member(s)			Publication date (day/month/year)
CN	104834590	A	12 August 2015	CN	104834590	B	23 November 2018
CN	105095087	A	25 November 2015		None		
US	2008178044	A1	24 July 2008	US	8533679	B2	10 September 2013