US 20160323397A1

(19) **United States**

(12) **Patent Application Publication**

Nagaraj et al.

(10) **Pub. No.: US 2016/0323397 A1**

(43) **Pub. Date: Nov. 3, 2016**

(54) **AYSNCHRONOUS CUSTOM EXIT POINTS**

(71) Applicant: **AppDynamics Inc.**, San Francisco, CA (US)

(72) Inventors: **Sanjay Nagaraj**, Dublin, CA (US); **Ryan Ericson**, San Francisco, CA (US); **Alex Fedotyev**, San Francisco, CA (US)

(21) Appl. No.: **14/701,418**

(22) Filed: **Apr. 30, 2015**

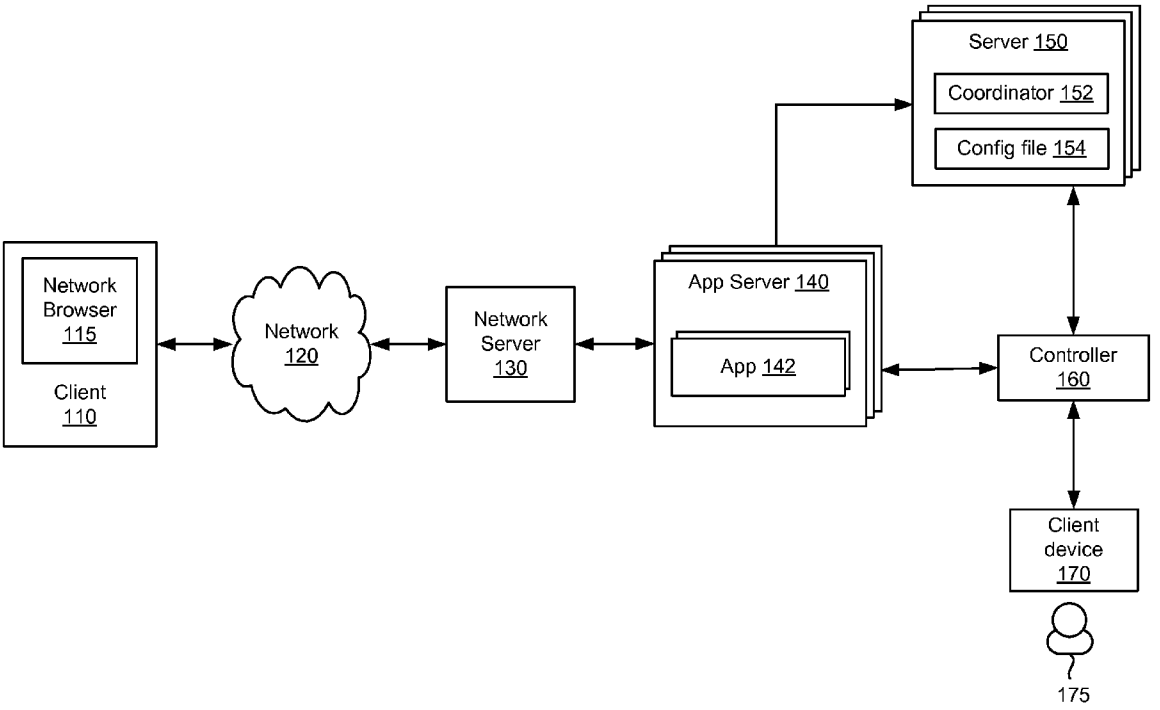
Publication Classification

(51) **Int. Cl.**
H04L 29/08 (2006.01)

(52) **U.S. Cl.**
CPC ***H04L 67/22*** (2013.01); ***H04L 67/02*** (2013.01)

(57) **ABSTRACT**

The present technology may monitor an asynchronous transaction based on a custom exit point. Once an asynchronous method to be monitored has been identified, the transition framework may be tracked while executing the asynchronous method call. Within a.NET framework, monitoring may include tracking a task object, continuation method calls at the completion of a method, and tracking the continuation method as it executes other code. The asynchronous method may then be correlated within a business transaction using the returned task object data.



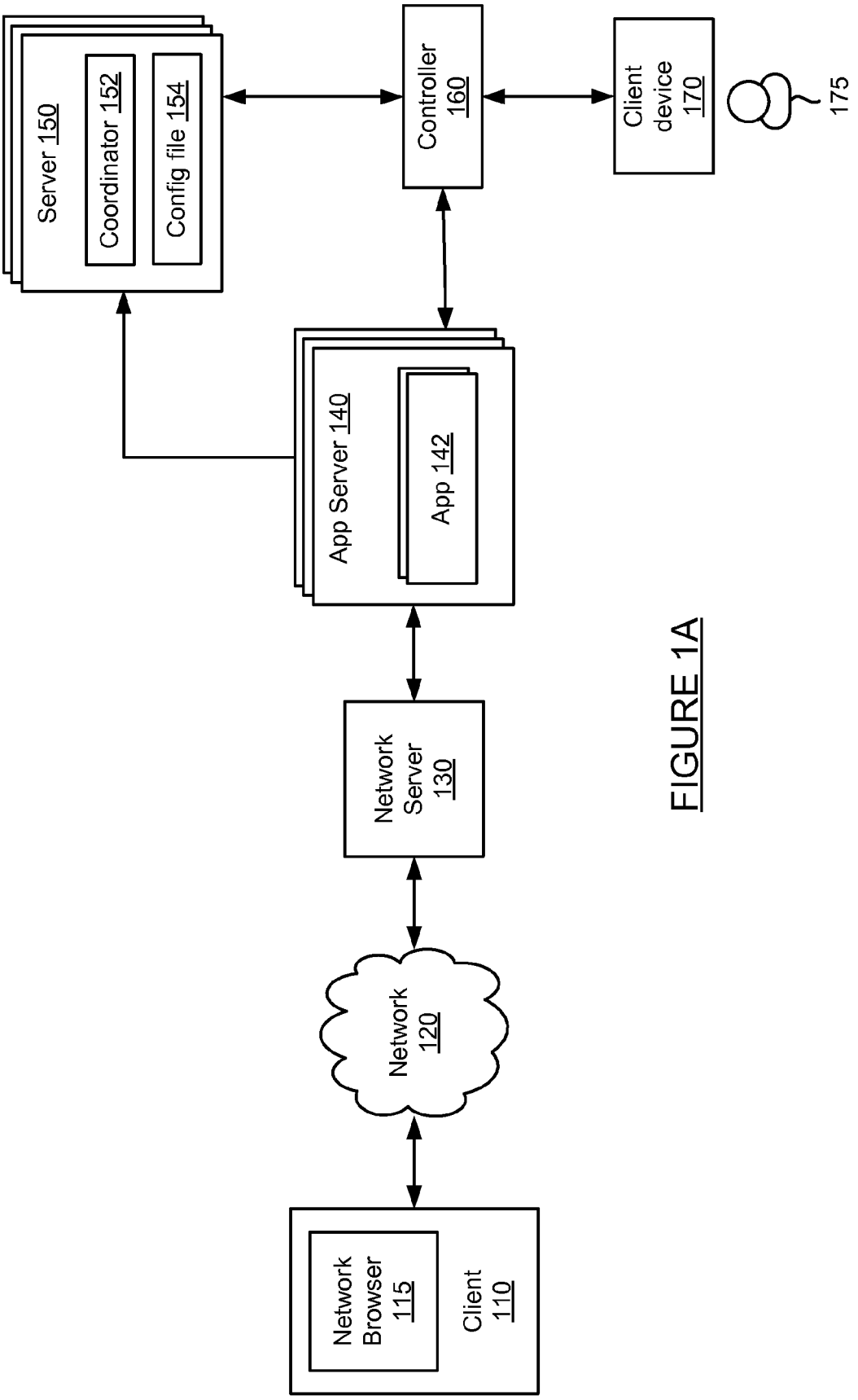

Server 150
Coordinator 152
Config file 154
Controller 160
Client device 170
175
App Server 140
App 142
Network Server 130
Network 120
Network Browser 115
Client 110

FIGURE 1A

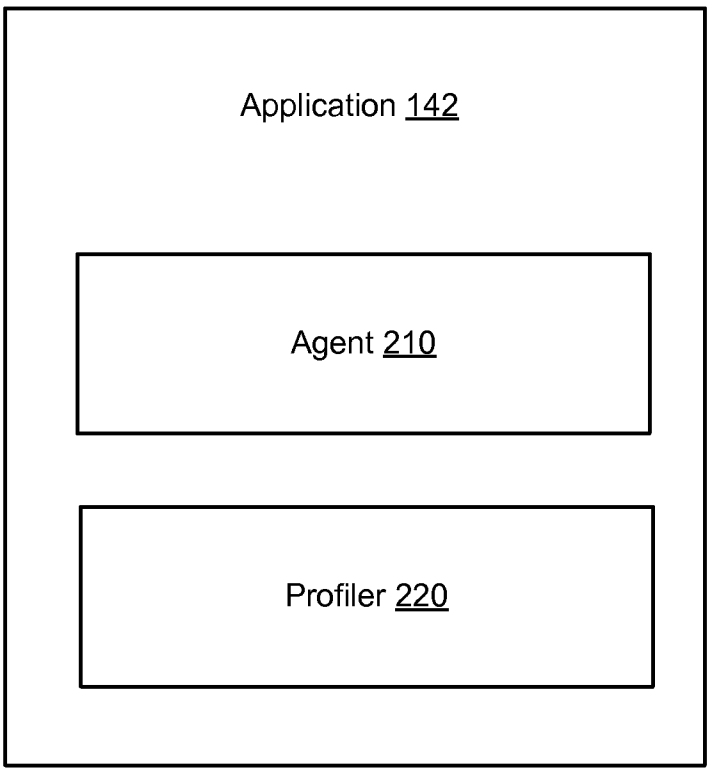


FIGURE 1B

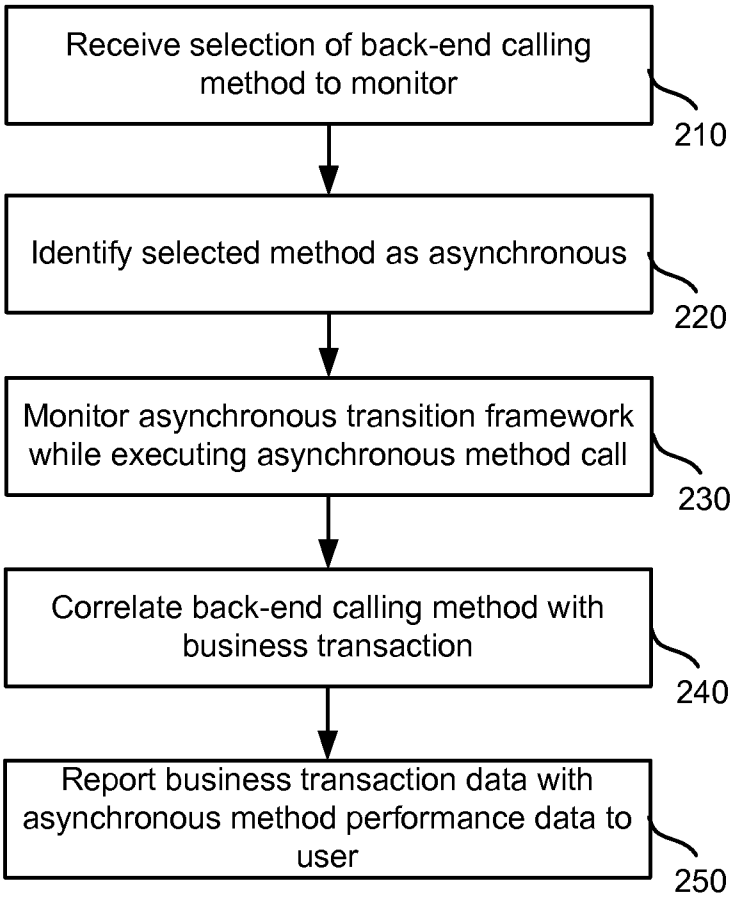

Receive selection of back-end calling method to monitor
210
Identify selected method as asynchronous
220
Monitor asynchronous transition framework while executing asynchronous method call
230
Correlate back-end calling method with business transaction
240
Report business transaction data with asynchronous method performance data to user
250

FIGURE 2

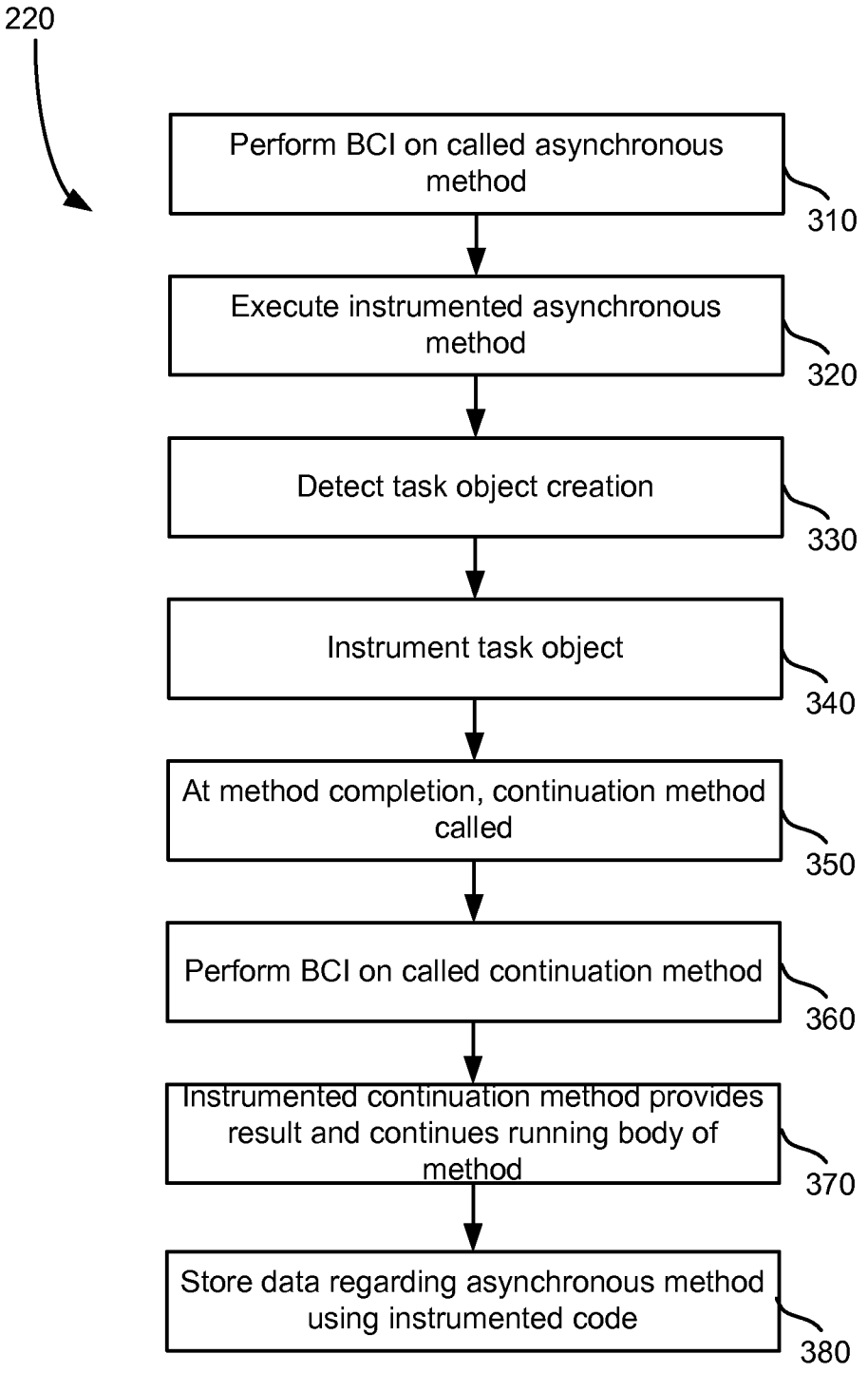

220
Perform BCI on called asynchronous method
310
Execute instrumented asynchronous method
320
Detect task object creation
330
Instrument task object
340
At method completion, continuation method called
350
Perform BCI on called continuation method
360
Instrumented continuation method provides result and continues running body of method
370
Store data regarding asynchronous method using instrumented code
380

FIGURE 3

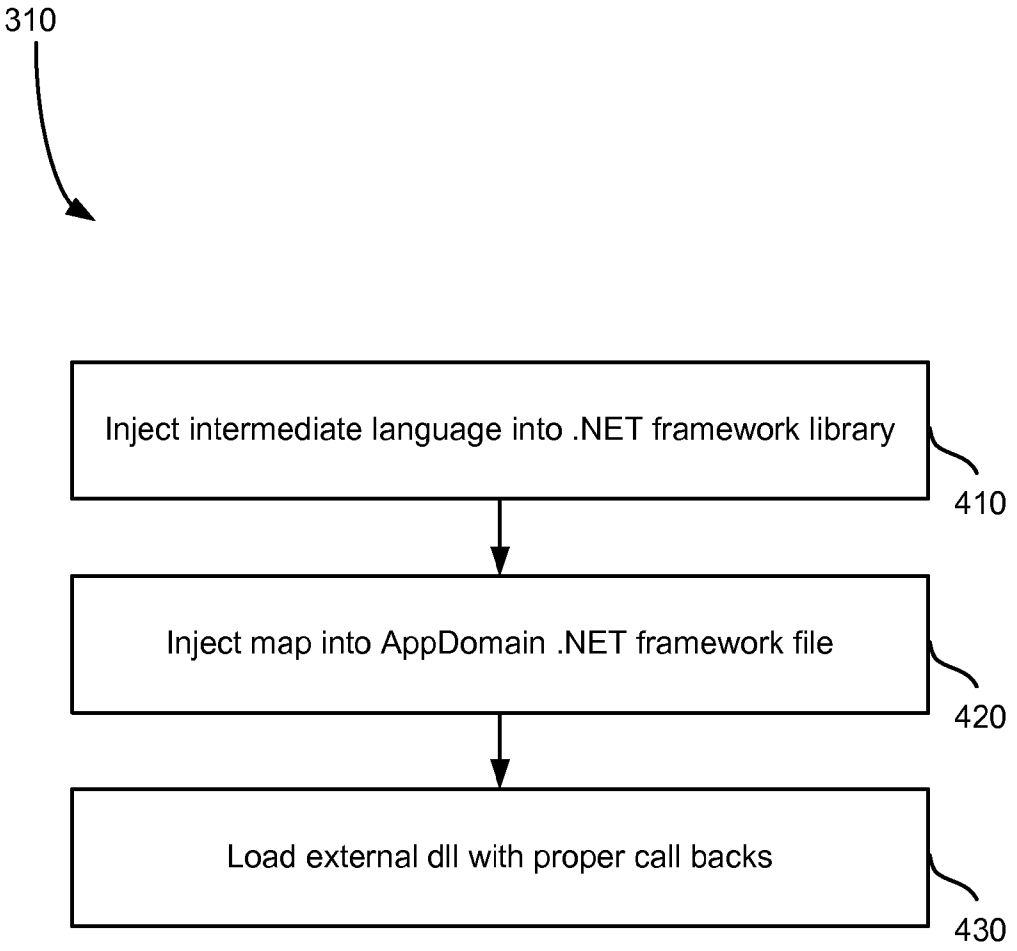

310
Inject intermediate language into .NET framework library
410
Inject map into AppDomain .NET framework file
420
Load external dll with proper call backs
430

FIGURE 4

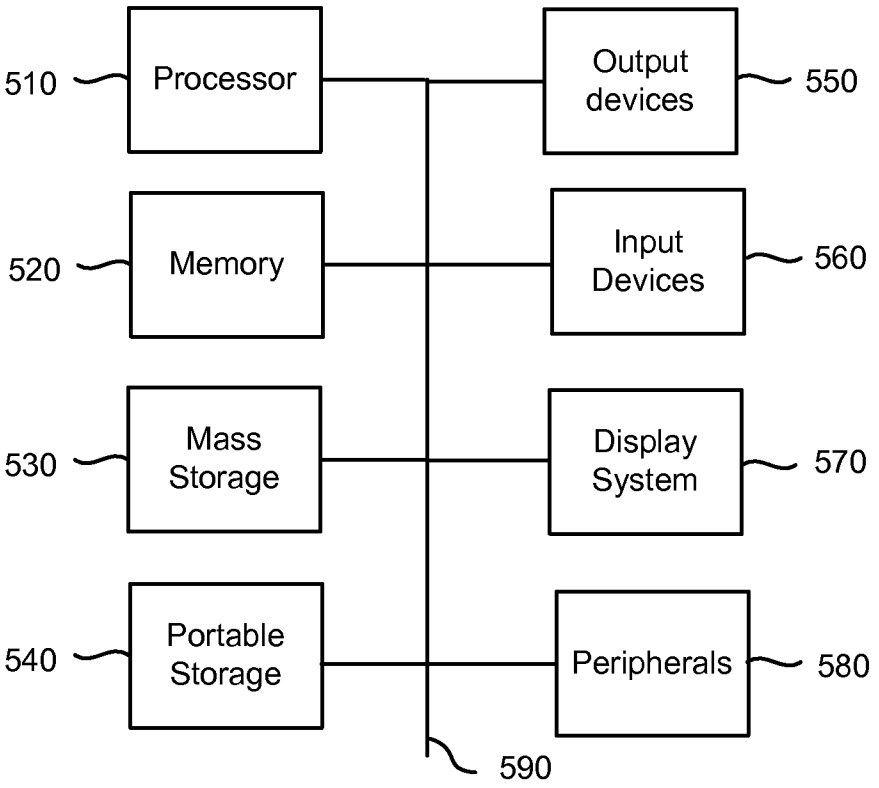

510
Processor
520
Memory
530
Mass Storage
540
Portable Storage
590
Output devices
550
Input Devices
560
Display System
570
Peripherals
580

FIGURE 5

AYSNCHRONOUS CUSTOM EXIT POINTS

BACKGROUND OF THE INVENTION

[0001] The World Wide Web has expanded to provide web services faster to consumers. Web services may be provided by a web application which uses one or more services to handle a transaction. The applications may be distributed over several machines, making the topology of the machines that provides the service more difficult to track and monitor.

[0002] A popular framework for providing a web is the .NET framework provided by Microsoft, Corp. In a .NET framework, certain transactions such as asynchronous transactions can be difficult to monitor. This is primarily due to the fact that a first thread may handle a first portion of a distributed business transaction, a second thread may handling another part of the distributed business transaction, and there is no connection or correlation between the two threads within the business transaction. What is needed is an improved manner for tracking asynchronous transactions.

SUMMARY OF THE CLAIMED INVENTION

[0003] The present technology may monitor an asynchronous transaction based on a custom exit point. Once an asynchronous method to be monitored has been identified, the transition framework may be tracked while executing the asynchronous method call. Within a.NET framework, monitoring may include tracking a task object, continuation method calls at the completion of a method, and tracking the continuation method as it executes other code. The asynchronous method may then be correlated within a business transaction using the returned task object data.

[0004] An embodiment may include a method for monitoring an asynchronous transaction. The method may detect an asynchronous method call within an application by an agent executing on a server. A task object associated with the method call may be monitored. The task object creation may be initiated by the asynchronous method call. Asynchronous method call data may be correlated with a distributed business transaction performed at least in part on the server.

[0005] An embodiment may include a system for monitoring a business transaction. The system may include a processor, a memory and one or more modules stored in memory and executable by the processor. When executed, the one or more modules may detect an asynchronous method call within an application by an agent executing on a server, monitor a task object associated with the method call, the task object creation initiated by the asynchronous method call, and correlate asynchronous method call data with a distributed business transaction performed at least in part on the server.

BRIEF DESCRIPTION OF THE DRAWINGS

[0006] FIG. **1**A is a block diagram of a system for monitoring an asynchronous transaction based on a custom exit point.

[0007] FIG. **1**B is a block diagram of a node.

[0008] FIG. **2** illustrates a method for monitoring and asynchronous transaction.

[0009] FIG. **3** illustrates a method for monitoring asynchronous transition framework.

[0010] FIG. **4** illustrates a method for instrumenting an asynchronous method.

[0011] FIG. **5** is a block diagram of a computing system implementing the present technology

DETAILED DESCRIPTION

[0012] The present technology may monitor an asynchronous transaction based on a custom exit point. Once an asynchronous method to be monitored has been identified, the transition framework may be tracked while executing the asynchronous method call. Within a.NET framework, monitoring may include tracking a task object, continuation method calls at the completion of a method, and tracking the continuation method as it executes other code. The asynchronous method may then be correlated within a business transaction using the returned task object data.

[0013] A .NET framework may include one or more IIS web servers as well as additional servers. Each IIS server may include one or more applications and at least one additional server may include a coordinator. An application being monitored on an IIS server may include an agent and/or a profiler. The profiler may detect a call within or by the application and report the call to the coordinator. The coordinator may determine if the detected call is one that should be monitored, and informs the profiler appropriately. If the call should be monitored, and agent on the application monitors the call. In some instances, more or fewer modules than an agent and profiler may be used to monitor an application on a .NET framework. References to an agent and profiler are intended for purposes of example only.

[0014] One aspect of the present technology is that the asynchronous framework of the .NET application is monitored. In a.NET framework, an asynchronous method may be called as a task object. The method may be compiled in the .NET framework with a C# compiler. The .NET framework compiler may create a state machine and replace an await function with code that sets a continuation method. The present technology may instrument selected asynchronous methods, continuation constructor methods, task objects, and other framework aspects. When the task completes, the continuation method is called. Understanding the.NET framework and instrumenting it as it progresses allows the present technology to track a custom exit point formed by an asynchronous method performed within that framework.

[0015] FIG. **1**A is a block diagram of a system for monitoring an asynchronous transaction based on a custom exit point. FIG. **1**A includes client **110**, network **120**, network server **130**, application server **140**, server **150**, controller **160**, and client device **170**. Client **110** may communicate with network server **130** over network **120**. Client **110** may be any sort of computing device, such as for example a desktop computer, a work station, a lap top computer a mobile device such as a smart phone or a tablet computer, or some other computing device. Client **110** may include network browser **115** as well as other software. Network browser **115** may be stored on client **110** and executed by one or more processors to provide content through an output device of client **110**. The content may be received from application server **140** via network server **130** and network **120**. Client **110** may receive input from a user through network browser **115** and communicate with application **1** server to provide content to the user.

[0016] Network **120** may facilitate communication of data between different servers, devices and machines. The network may be implemented as a private network, public

network, intranet, the Internet, a Wi-Fi network, cellular network, or a combination of these networks.

[0017] Network server **130** is connected to network **120** and may receive and process requests received over network **120**. Network server **130** may be implemented as one or more servers implementing a network service. When network **120** is the Internet, network server **125** may be implemented as a web server. Network server **130** and application server **140** may be implemented on separate or the same server or machine.

[0018] Application server **140** may include one or more applications **142**. Application server **140** may be implemented using one or more servers which communicate with network server **130**, server **150**, controller **160**, and other devices. In some embodiments, network server **130** and application server **140** may be implemented as the same server.

[0019] Application **142** may be monitored by one or more agents (see FIG. **1**B). Application **142** may execute in any of a number of frameworks, such as for example a JAVA framework, a .NET framework, or other framework. Application **142** is discussed in more detail below with respect to the method of FIG. **1**B.

[0020] Server **150** may communicate with application servers **140** and controller **160**. Server **150** may include a coordinator **152** and a configuration file **154**. Coordinator **152** may manage a list of methods, calls, objects and other code that should be monitored. Configuration file **154** may be accessed by coordinator **152** and may include a list of nodes that may be monitored within the system of FIG. **1**A. The list of nodes may be compiled automatically, based on user input, or based on other parameters.

[0021] Controller **160** may control and manage monitoring of business transactions distributed over application servers **130**-**160**. Controller **160** may receive runtime data from agents and coordinators, associate portions of business transaction data, communicate with agents to configure collection of runtime data, and provide performance data and reporting through an interface. The interface may be viewed as a web-based interface viewable by client device **110**. In some embodiments, a client device **170** may directly communicate with controller **160** to view an interface for monitoring data.

[0022] In some instances, controller **160** may install an agent into one or more application servers **130**. Controller **160** may receive correlation configuration data, such as an object, a method, or class identifier, from a user through client device **192**.

[0023] FIG. **1**B is a block diagram of an application. Application **200** of FIG. **1**B includes agent **210** and profiler **220**.

[0024] Agent **210** may be installed on an application server by byte code instrumentation, downloading the application to the server, or in some other manner. Agent **210** may be executed to monitor an application, application server, a virtual machine, or other logical machine and may communicate with byte instrumented code on an application server, virtual machine **132** or another application or program on an application server. Agent **210** may detect operations such as receiving calls, creating objects, and sending requests by an application server, virtual machine, or logical machine. Agent **210** may insert instrumentation and receive data from instrumented code, process the data and transmit the data to controller **190**. Agent **210** may perform other operations related to monitoring an application or logical machine as discussed herein. For example, agent **210** may identify other applications, share business transaction data, aggregate detected runtime data, and other operations.

[0025] Profiler **220** may detect when an application makes a call and may take actions based on that detection. Profiler **220** may be implemented within or outside of application **200** on an application server. Profiler **220** may communicate with coordinator **152** to determine if the application making the call should be monitored. In some instances, profiler **220** may implement byte code to activate an agent or cause an agent to be activated in case that the application should be monitored based on information received from coordinator **152**.

[0026] FIG. **2** illustrates a method for monitoring an asynchronous transaction. A selection of a back-end calling method to monitor is received at step **210**. The method that calls the back-end may provide an exit point from the current application to the external back-end. Subsequent monitoring of the selected method provides custom back-end monitoring within the current framework. The method may be selected based on user request, automated selection, or other means.

[0027] The selected method may be identified as an asynchronous method as step **220**. In some instances, the present system may automatically determine if the selected method is an asynchronous method. For example, the system may detect a type "task" within the method, which in a .NET framework corresponds to an asynchronous method.

[0028] Next, the asynchronous transaction framework is monitored while executing the asynchronous method call at step **230**. Monitoring the asynchronous transaction framework may include detecting operations of the framework, such as executing the asynchronous method, instrumenting asynchronous methods, instrumenting a continuation method, and other monitoring operations. More details for monitoring and asynchronous transaction framework are discussed with respect to the method of FIG. **3**.

[0029] The back-end calling method is correlated with a business transaction at step **230**. A first thread may be handling the asynchronous method identified at step **210**. A second method may handle a task object that is executed as part of the asynchronous method. Data obtained from monitoring the asynchronous method and task object may be used to correlate the threads together as part of a distributed business transaction. More details for correlating are discussed with respect to the method of FIG. **4**. Data regarding the performance of the business transaction is reported at step **240**. The data may include asynchronous method call information as part of the end to end business transaction. The data may be reported as part of a call graph, trending data, graphics, and other means.

[0030] FIG. **3** illustrates a method for monitoring an asynchronous transition framework. The method of FIG. **3** provides more detail for step **220** of the method of FIG. **2**. First, BCI is performed on the called asynchronous method at step **310**. The instrumentation may include creating a new method that encompasses the asynchronous method to be monitored within a wrapper. The wrapper may include code for monitoring the start and end of the called method, collecting data for the method, and other monitoring actions. The instrumentation itself may include modifying a frame-

work library and files. More detail for instrumenting the asynchronous method is discussed with respect to the method of FIG. **4**.

[0031] Next, the instrumented asynchronous method is executed at step **320**. A task object creation is detected at step **330** and the task object is instrumented at step **340**. Task object instrumentation is used to correlate different threads associated with the asynchronous business transaction. For example, when the task object sends a call to the back-end, an identifier may be provided in the call. The identifier may be stored in the thread handling the asynchronous method in the .NET application, any thread used to process the completion of the call, and a thread or other code at the back-end server. The identifier may be used to correlate the threads involved with aspects of the asynchronous method on the .NET application as well as the remote back-end server.

[0032] At the completion of the synchronous method, a continuation method is called at step **350**. The continuation method is instrumented at step **360**. Instrumentation for the continuation method may result in data such as the time the continuation method is called, the recipient of the continuation method call, and other data. The instrumented continuation method may also provide results of the task object and details regarding code in the body of the asynchronous method which is executed by the continuation method after the task is complete at step **370**. Data regarding the asynchronous method is then stored using the instrumented code at step **380**.

[0033] FIG. **4** illustrates a method for instrumenting an asynchronous method. The method of FIG. **4** provides more detail for step **310** the method of FIG. **3**.

[0034] Intermediate language (IL) is injected into a .NET framework library at step **410**. The library may be an mscorlib.dll library within the .NET framework. The code inserted may include a code at the beginning and end of the asynchronous method, thereby providing a wrapper around the method.

[0035] A map is inserted into an AppDomain file at step **420**. The map indicates a method begin pointer and method end pointer, both of which are expected to be part of the app domain. The AppDomain file may be part of the .NET framework and serve as a placeholder method.

[0036] An external dll is loaded with proper call backs at step **430**. The proper call backs are used to fill the placeholder with the proper call back information. The map may be accessed with information regarding mapping call backs to a specific method.

[0037] FIG. **5** is a block diagram of a computer system for implementing the present technology. System **500** of FIG. **5** may be implemented in the contexts of the likes of clients **110** and **170**, network server **130**, servers **140-150**, and controller **160**.

[0038] The computing system **500** of FIG. **5** includes one or more processors **510** and memory **520**. Main memory **520** stores, in part, instructions and data for execution by processor **510**. Main memory **510** can store the executable code when in operation. The system **500** of FIG. **5** further includes a mass storage device **530**, portable storage medium drive(s) **540**, output devices **550**, user input devices **560**, a graphics display **570**, and peripheral devices **580**.

[0039] The components shown in FIG. **5** are depicted as being connected via a single bus **590**. However, the components may be connected through one or more data transport means. For example, processor unit **510** and main memory **520** may be connected via a local microprocessor bus, and the mass storage device **530**, peripheral device(s) **580**, portable storage device **540**, and display system **570** may be connected via one or more input/output (I/O) buses.

[0040] Mass storage device **530**, which may be implemented with a magnetic disk drive or an optical disk drive, is a non-volatile storage device for storing data and instructions for use by processor unit **510**. Mass storage device **530** can store the system software for implementing embodiments of the present invention for purposes of loading that software into main memory **520**.

[0041] Portable storage device **540** operates in conjunction with a portable non-volatile storage medium, such as a floppy disk, compact disk or Digital video disc, to input and output data and code to and from the computer system **500** of FIG. **5**. The system software for implementing embodiments of the present invention may be stored on such a portable medium and input to the computer system **500** via the portable storage device **540**.

[0042] Input devices **560** provide a portion of a user interface. Input devices **560** may include an alpha-numeric keypad, such as a keyboard, for inputting alpha-numeric and other information, or a pointing device, such as a mouse, a trackball, stylus, or cursor direction keys. Additionally, the system **500** as shown in FIG. **5** includes output devices **550**. Examples of suitable output devices include speakers, printers, network interfaces, and monitors.

[0043] Display system **570** may include a liquid crystal display (LCD) or other suitable display device. Display system **570** receives textual and graphical information, and processes the information for output to the display device.

[0044] Peripherals **580** may include any type of computer support device to add additional functionality to the computer system. For example, peripheral device(s) **580** may include a modem or a router.

[0045] The components contained in the computer system **500** of FIG. **5** are those typically found in computer systems that may be suitable for use with embodiments of the present invention and are intended to represent a broad category of such computer components that are well known in the art. Thus, the computer system **500** of FIG. **5** can be a personal computer, hand held computing device, telephone, mobile computing device, workstation, server, minicomputer, mainframe computer, or any other computing device. The computer can also include different bus configurations, networked platforms, multi-processor platforms, etc. Various operating systems can be used including Unix, Linux, Windows, Macintosh OS, Palm OS, Android OS, and other suitable operating systems.

[0046] When implementing a mobile device such as smart phone or tablet computer, the computer system **500** of FIG. **5** may include one or more antennas, radios, and other circuitry for communicating over wireless signals, such as for example communication using Wi-Fi, cellular, or other wireless signals.

[0047] The foregoing detailed description of the technology herein has been presented for purposes of illustration and description. It is not intended to be exhaustive or to limit the technology to the precise form disclosed. Many modifications and variations are possible in light of the above teaching. The described embodiments were chosen in order to best explain the principles of the technology and its practical application to thereby enable others skilled in the art to best utilize the technology in various embodiments and

with various modifications as are suited to the particular use contemplated. It is intended that the scope of the technology be defined by the claims appended hereto.

What is claimed is:

1. A method for monitoring an asynchronous transaction, comprising:

detecting an asynchronous method call within an application by an agent executing on a server;

monitoring a task object associated with the method call, the task object creation initiated by the asynchronous method call; and

correlating asynchronous method call data with a distributed business transaction performed at least in part on the server.

2. The method of claim **1**, wherein tracking includes performing byte code instrumentation on a called asynchronous method.

3. The method of claim **1**, wherein instrumenting includes injecting intermediate language and a map into a .NET framework.

4. The method of claim **1**, wherein tracking includes performing byte code instrumentation on a task object associated with the asynchronous method.

5. The method of claim **1**, further comprising:

receiving a selection of a back-end calling method to monitor; and

automatically identifying the selected method as asynchronous.

6. The method of claim **1**, further comprising reporting performance of a distributed business transaction that includes a call to the asynchronous method.

7. The method of claim **1**, wherein the framework is a .NET framework.

8. A non-transitory computer readable storage medium having embodied thereon a program, the program being executable by a processor to perform a method for monitoring an asynchronous transaction, the method comprising:

detecting an asynchronous method call within an application by an agent executing on a server;

monitoring a task object associated with the method call, the task object creation initiated by the asynchronous method call; and

correlating asynchronous method call data with a distributed business transaction performed at least in part on the server.

9. The non-transitory computer readable storage medium of claim **7**, wherein tracking includes performing byte code instrumentation on a called asynchronous method.

10. The non-transitory computer readable storage medium of claim **7**, wherein instrumenting includes injecting intermediate language and a map into a .NET framework.

11. The non-transitory computer readable storage medium of claim **7**, wherein tracking includes performing byte code instrumentation on a task object associated with the asynchronous method.

12. The non-transitory computer readable storage medium of claim **7**, the method further comprising:

receiving a selection of a back-end calling method to monitor; and

automatically identifying the selected method as asynchronous.

13. The non-transitory computer readable storage medium of claim **7**, the method further comprising reporting performance of a distributed business transaction that includes a call to the asynchronous method.

14. The non-transitory computer readable storage medium of claim **7**, wherein the framework is a .NET framework.

15. A system for monitoring a business transaction, comprising:

a processor;

a memory; and

one or more modules stored in memory and executable by a processor to detect an asynchronous method call within an application by an agent executing on a server, monitor a task object associated with the method call, the task object creation initiated by the asynchronous method call, and correlate asynchronous method call data with a distributed business transaction performed at least in part on the server.

16. The system of claim **15**, wherein tracking includes performing byte code instrumentation on a called asynchronous method.

17. The system of claim **15**, wherein instrumenting includes injecting intermediate language and a map into a .NET framework.

18. The system of claim **15**, wherein tracking includes performing byte code instrumentation on a task object associated with the asynchronous method.

19. The system of claim **15**, the one or more modules further executable to receive a selection of a back-end calling method to monitor and automatically identify the selected method as asynchronous.

20. The system of claim **15**, the one or more modules further executable to report performance of a distributed business transaction that includes a call to the asynchronous method.

21. The system of claim **15**, wherein the framework is a .NET framework.

* * * * *