END-TO-END TRANSACTION TRACKING IN THE ENTERPRISE

Inventor: Jean-Jacques Heler, Palo Alto, CA (US)

Abstract

Various embodiments of a method, system, and article of manufacture track the processing of a transaction. A managing server receives an originator communication event record from a first collector associated with a first application environment having a first unit of execution. The first communication event record is associated with a first unit of execution identifier and a first token. The managing server receives a second communication event record from a second collector associated with a second application environment having a second unit of execution. The second communication event record is associated with a second unit of execution identifier and a second token. The managing server determines whether the first unit of execution and the second unit of execution are associated with processing a same transaction based on the first token and the second token.
FIG. 2

<table>
<thead>
<tr>
<th>UOE identifier</th>
<th>Timestamp</th>
</tr>
</thead>
</table>

FIG. 3A

<table>
<thead>
<tr>
<th>Hosting thread identifier</th>
<th>Timestamp</th>
</tr>
</thead>
</table>

FIG. 3B

<table>
<thead>
<tr>
<th>TCA number</th>
<th>Timestamp</th>
</tr>
</thead>
</table>

FIG. 4

<table>
<thead>
<tr>
<th>AE identifier</th>
<th>Trip counter</th>
</tr>
</thead>
</table>

FIG. 5

<table>
<thead>
<tr>
<th>Message identifier</th>
<th>Correlation identifier</th>
</tr>
</thead>
</table>

FIG. 6

<table>
<thead>
<tr>
<th>Correlation identifier</th>
<th>Message type</th>
</tr>
</thead>
</table>
Trap, at an originating application environment, an attempt to communicate by the originating UOE executing in an originating application environment with a receiving UOE in a receiving application environment, and activate a collector associated with the originating application environment.

Generate, by the collector, a token associated with the originating UOE.

Send, by the collector, an originator communication event record to the managing server, the originator communication event record comprising an event type, an originating UOE identifier, the token and a timestamp.

Exit the collector.

Send the communication, with the token, to the receiving UOE.

FIG. 7

Trap, at the receiving application environment, the receipt of a communication from an originating UOE in an originating application environment, by a receiving UOE, the communication comprising a token, and activate the collector associated with the receiving application environment.

Extract, by the collector, the token from the communication.

Send, by the collector, a receiver communication event record to the managing server, the receiver communication event record comprising an event type, a receiving UOE identifier, the token and the timestamp.

Exit the collector.

FIG. 8
Trap, at the receiving application environment, an attempt to send a communication which indicates that the receiving UOE is suspending or stopping execution, to the originating UOE, and activate the collector associated with the receiving application environment.

Retrieve, by the collector, the token which was received in the communication from the originating UOE to provide a received token.

Store, by the collector, the received token in the communications structure associated with the communication which indicates that the receiving UOE is suspending or stopping execution.

Send, by the collector, a receiver communication event record to the managing server, the receiver communication event record comprising an event type, the receiving UOE identifier, the received token and a timestamp.

Exit the collector.

Send the communication which indicates that the receiving UOE is suspending or stopping execution, to the originating UOE with the received token.

FIG. 9

Trap, at the originating application environment, the receipt of a communication that execution of a receiving UOE has been suspended or stopped, the communication comprising a token, and activate the collector associated with originating application environment.

Extract, by the collector, the token from the communication.

Send, by the collector, an originating communication event record to the managing server, the originating communication event record comprising an event type, the receiving UOE identifier, the token and the timestamp.

Exit the collector.

FIG. 10
172 Trap, at the originating application environment, the return from an API to send a communication to the receiving UOE using a messaging system, and activate a collector associated with the originating application environment.

174 Extract, by the collector, the token from the communication.

176 Send, by the collector, an originating communication event record to the managing server, the originating communication event record comprising an event type, the originating UOE identifier, the token and a timestamp.

178 Exit the collector.

FIG. 11

180 Trap, at the receiving application environment, the return from an API to retrieve a communication, and activate a collector associated with the receiving application environment.

182 Extract, by the collector, the token from the communication.

184 Send, by the collector, a receiver communication event record to the managing server, the receiver communication event record comprising an event type, the receiving UOE identifier, the token and a timestamp.

186 Exit the collector.

FIG. 12
Receive and store originator and receiver communication event records, each communication event record having a token.

Correlate UOE1s across application environments based on the token in the originator and receiver communication event records.

Provide a visual indication of the correlation between the UOE1s across application environments.

FIG. 13

FIG. 14
FIG. 15
FIG. 18
END-TO-END TRANSACTION TRACKING IN THE ENTERPRISE

BACKGROUND OF THE INVENTION

[0001] 1.0 Field of the Invention

[0002] This invention relates to tracking transactions; and in particular, to tracking individual transactions as they are processed across various application environments.

[0003] 2.0 Description of the Related Art

[0004] With the advent of the Internet and Web computing, enterprises have had to expand their information technology (IT) to a more distributed environment, while keeping most of their business logic in a traditional mainframe environment using legacy software systems. Exemplary legacy software systems comprise IBM® (Registered trademark of International Business Machines Corporation) Customer Information Control System (CICS® (Registered trademark of International Business Machines Corporation)), IBM Information Management System (IMSTM (Trademark of International Business Machines Corporation)), and DB2® (Registered trademark of International Business Machines Corporation). The distributed environment has increased the need for software systems such as Web servers and application servers, for example, Java® 2 Platform Enterprise Edition (J2EE® (Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both)) servers, to satisfy on-line access through the World Wide Web. However, transactions are still processed through the legacy software systems, using synchronous gateways or asynchronous messaging systems such as IBM WebSphere® (Registered trademark of International Business Machines Corporation) MQSeries® (Registered trademark of International Business Machines Corporation). A legacy software system is a non-J2EE software system. As a result, the processing of a business transaction is rarely confined to one application server or system. The processing of the business transaction typically flows from a Web server to an application server all the way to the legacy software system and back. In addition, the advent of the Internet has greatly increased the volume of such transactions twenty-four hours a day, seven days a week. IT organizations need to provide an acceptable end-to-end response time for processing transactions. Therefore, there is a need for IT organizations to track the processing of transactions should problems such as bottlenecks, deadlocks and hangs occur.

[0005] Traditional solutions have typically consisted of separately monitoring a software system either from the resource point-of-view or from the application point-of-view to provide information regarding the behavior of transactions while the transactions are being processed by that software system. Monitoring from the resource point-of-view comprises monitoring the memory utilization, central processing unit (CPU) time, and connection pools of the software system. Monitoring from the application point-of-view comprises monitoring transaction performance metrics, such as response time, or tracing the execution of a single instance.

[0006] Some conventional monitors only have a view of the specific software system that they are monitoring and are not able to correlate the processing across various software systems. A servlet is a Java program for providing additional features to a Web server and can be used to access existing business systems. One conventional J2EE monitor is able to identify the servlets which have requested external services from CICS. Another conventional CICS monitor is able to detect activities in CICS. However, neither the J2EE monitor nor the CICS monitor is able to associate any servlet with any CICS tasks that the servlet has invoked or sent messages to. When a problem occurs, it becomes difficult, if not impossible, to identify the problem or know what action to take in order to correct the problem. Therefore there is a need for an improved technique which allows the processing of transactions to be tracked across various software systems.

SUMMARY OF THE INVENTION

[0007] To overcome the limitations in the prior art described above, and to overcome other limitations that will become apparent upon reading and understanding the present specification, various embodiments of a method, system and article of manufacture which track the processing of a transaction are provided. A managing server receives an originator communication event record from a first collector associated with a first application environment having a first unit of execution. The first communication event record comprises a first unit of execution identifier and a first token. The managing server receives a second communication event record from a second collector associated with a second application environment having a second unit of execution. The second communication event record comprises a second unit of execution identifier and a second token. The managing server determines whether the first unit of execution and the second unit of execution are associated with processing a transaction based on the first token and the second token. In some embodiments, the managing server determines that the first unit of execution and the second unit of execution are associated with the processing of the same transaction in response to the first token matching the second token.

[0008] In this way, the processing of transactions can be tracked across various units of execution in different application environments, and therefore, the processing of transactions can be tracked across various software systems.

BRIEF DESCRIPTION OF THE DRAWINGS

[0009] The teachings of the present invention can be readily understood by considering the following description in conjunction with the accompanying drawings, in which:

[0010] FIG. 1 depicts a diagram illustrating a managing server and application environments;

[0011] FIG. 2 depicts an embodiment of a unit of execution (UOE) tag comprising a UOE identifier and a timestamp;

[0012] FIG. 3A depicts a diagram of an embodiment of a UOE tag comprising a hosting thread identifier as the UOE identifier, and a timestamp;

[0013] FIG. 3B depicts a diagram of another embodiment of a UOE tag comprising a task control area number as the UOE identifier, and a timestamp;

[0014] FIG. 4 depicts a diagram of an embodiment of a native token;
FIG. 5 depicts a diagram of an embodiment of a guest token;

FIG. 6 depicts a diagram of another embodiment of a guest token;

FIG. 7 depicts a flowchart of an embodiment of sending an originator communication event record by a collector associated with an originating application environment in response to an attempt to communicate with a receiving UOE in a receiving application environment;

FIG. 8 depicts a flowchart of an embodiment of sending a receiver communication event record by a collector associated with a receiving UOE in a receiving application environment in response to receiving a communication from an originating UOE in an originating application environment;

FIG. 9 depicts a flowchart of an embodiment of sending a receiver communication event record by a collector associated with the receiving application environment in response to the receiving UOE attempting to send a communication which indicates that the receiving UOE is suspending or stopping execution;

FIG. 10 depicts a flowchart of an embodiment of sending an originator communication event record by a collector associated with an originating application environment in response to a recipient sending a message using a message queue;

FIG. 11 depicts a flowchart of an embodiment of sending an originator communication event record by a collector associated with an originating application environment in response to an asynchronous sending of a message using a message queue;

FIG. 12 depicts a flowchart of an embodiment of sending a receiver communication event record by a collector associated with a receiving application environment in response to the asynchronous receiving of a message from a message queue;

FIG. 13 depicts a flowchart of an embodiment of the correlation of receiver and originator communication event records by a managing server;

FIG. 14 depicts a sequence diagram illustrating an embodiment of tracking the processing of a transaction across application environments using synchronous communication;

FIG. 15 depicts a sequence diagram illustrating an embodiment of tracking the processing of a transaction across application environments using non-message driven asynchronous and synchronous communication;

FIG. 16 depicts a sequence diagram illustrating an embodiment of tracking the processing of a transaction across application environments using message driven asynchronous communication;

FIG. 17 comprises FIGS. 17A, 17B and 17C which collectively depict an embodiment of a graphical user interface, generated by a global publishing server, which illustrates an exemplary trace of the processing of a transaction across various units of execution in different application environments; and

FIG. 18 depicts an illustrative computer system which uses various embodiments of the present invention.

To facilitate understanding, identical reference numerals have been used, where possible, to designate identical elements that are common to some of the figures.

DETAILED DESCRIPTION

After considering the following description, those skilled in the art will clearly realize that the teachings of the various embodiments of the present invention can be utilized to track the processing of a transaction. A managing server receives an originator communication event record from a first collector associated with a first application environment having a first unit of execution. The first communication event record comprises a first unit of execution identifier and a first token. The managing server receives a second communication event record from a second collector associated with a second application environment having a second unit of execution. The second communication event record comprises a second unit of execution identifier and a second token. The managing server determines whether the first unit of execution and the second unit of execution are associated with processing the same transaction based on the first token and the second token. In some embodiments, the managing server determines that the first unit of execution and the second unit of execution are associated with the processing of the same transaction when the first token matches the second token.

FIG. 1 depicts a diagram illustrating a managing server 20 and application environments (AE) 32, 34 and 36. The managing server 24 is in computer system 22. Application environments 32 and 34 are in another computer system 24. Application environment 36 is in yet another computer system 26. In some embodiments, the managing server 20 also has at least one application environment. An “application environment” is a software environment in which applications are executed. In various embodiments, a “region” is an address space, that is, a portion of memory. In some embodiments, a “region” is a process, for example, a Unix (Unix is a registered trademark of the Open Group in the United States and other countries) process. In various embodiments, an application environment 32 and 34 comprises a plurality of regions 40 and 41. An application environment 32, 34 and 36 is associated with a collector 62, 64 and 66 which monitors the application environment 32, 34 and 36, respectively.

Examples of a single region application environment 38 and 39 include, and are not limited to, a CICS region and a distributed J2EE application server. A CICS region is an area of memory of the computer system in which CICS system programs have been loaded (and executed), with other memory allocated for the CICS region to use. The CICS region comprises a named collection of resources controlled by CICS as a unit. The resources in a CICS region comprise programs, Basic Mapping Support map sets, transactions, terminals, files, transient data queues, temporary storage queues and journals. In the distributed J2EE application server environment, also known as the application server instance, the application environment comprises one address space (or process or Java Virtual Machine (JVM)).
In another example, in some embodiments, an application environment comprises a plurality of regions. Examples of multi-region application environments include, and are not limited to, an IBM WebSphere Application Server (WAS/z) and IMS. In the WAS/z case, one application server instance has one control region and one or more JVMs, each JVM occupying one server region. In another example, one IMS application environment has one control region and one or more dependent regions, where the IMS applications are executed. In the WAS/z case a single collector is associated with the servant regions of the application environment; and, the control region is not associated with a collector. On the other hand, in the IMS case a single collector is associated with the control region; and, the dependent regions report to that collector via MVS® (Registered Trademark of International Business Machines Corporation) operating system cross memory-services.

A region 38, 39, 40 and 41 comprises at least one unit of execution (UOE) 42-44, 46-48, 50-52, and 54-56, respectively. In general, a unit of execution is a group of instructions which are executed as a unit, and have a beginning and an end. A UOE has a host mechanism for execution and a unique existence in time, for example, an instance. In various embodiments, the host mechanism for execution is a process, a thread, an operating system task control block (TCB) (MVS operating system, z/OS® (Registered Trademark of International Business Machines Corporation) operating system), a task control area (TCA) (CICS) or task (MVS operating system, z/OS operating system, CICS, IMS).

In various embodiments, a transaction is a set of input data and output data with a beginning and an end point in time, originated by a request and terminated when a response is returned to that request. The request can be a message or an invocation, depending on the embodiment. A transaction is executed in one or more application environments. A transaction which is executed in a single application environment is referred to as a local transaction (“local” to that particular application environment). A transaction that is executed in multiple application environments is referred to as a composite transaction. Therefore a composite transaction is composed of a plurality of local transactions. A composite transaction may also be referred to as a business transaction.

In general, a UOE executes in an application environment, and hosts a local transaction in that application environment. In various embodiments, the local transaction is executed as a sequence of one or more programs (CICS, methods (Java), functions (C/C++), or subroutines (COBOL, PL/I) in the application environment. In some embodiments, a UOE hosts one local transaction. In other embodiments, a UOE hosts more than one local transaction, for example, a server (as in the client-server architecture). In various embodiments, a server UOE is a long executing task which “serves” multiple external requests, each request resulting in a single local transaction; in these embodiments, an external request is from a different application environment from the application environment in which the server UOE is executing. An internal request is originated from the same application environment as the server UOE. The server UOE exists, that is, is executing, before the initiation of each local transaction that the server UOE hosts and exists after the completion of each local transaction that the server UOE hosts. If a UOE does not exist prior to and after completion of the local transaction, then the UOE hosts a single transaction.

In various embodiments, a UOE comprises a set of one or more programs, subroutines, functions or methods, which host one or more transactions. In some embodiments, the host mechanism for execution of a UOE is a thread. A thread is either in an active state or a “dormant” state and is identified by a thread identifier (id). A thread remains “dormant” in a thread pool in a process until the thread is associated with a UOE, in which case the thread becomes an active thread hosting a UOE. When the UOE terminates, the thread is no longer active and is returned to the thread pool and remains in the thread pool under the same thread id, until the thread is associated with another UOE. The UOE hosts transactions, whereas the thread is the software mechanism under which programs are executed. Hence, an executing thread “hosts” a UOE (or is referred to as being associated with a UOE). In other embodiments, the host mechanism for execution is a TCB, TCA or task, rather than a thread. However, the host mechanism for execution is not limited to a thread, TCB, TCA or task; and the invention will work with other embodiments of a host mechanism for execution.

Therefore, an application environment comprises one or more regions. In various embodiments, a region is a host to one or more threads or tasks or TCBS or TCAs, which typically remain in a pool ready to be associated with a UOE. A UOE executes in a region as an active thread, task, TCB, or TCA, depending on the embodiment. In order to execute, a UOE is associated with an active thread, task, TCB, or TCA, depending on the embodiment, which is said to host the UOE. In various embodiments, a UOE hosts one or more transactions. During its lifetime, the UOE can host one transaction, or the UOE can host more than one transaction.

In some embodiments, during execution, a UOE can originate or instantiate zero or more transactions. The transactions may be local or composite transactions. In various embodiments, a unit of execution may cause one or more other units of execution in other application environments to be executed.

In the MVS and z/OS operating systems, a TCB represents an operating system task, and there is at least one TCB per address space (the main TCB).

A CICS region is an address space, that is, an application environment, with several TCB’s. One specific TCB will execute the client requests which will take turns using that TCB in a round robin fashion. Each of these client requests, which are also referred to as CICS transactions, is identified by a CICS task number (not an operating system task), for which the information is to be found in the task control area (TCA). A CICS task is also referred to as a TCA, the same way as an MVS or z/OS operating system task is also referred to as a TCB. In various embodiments, a CICS transaction is identified by its task number, that is, its TCA number.

An IMS transaction is typically a local transaction which comprises a group of TCBs in different address spaces in a single IMS region.

In some embodiments, a Unix process is a UOE of an application environment which contains multiple UOEs;
or alternately, a Unix process can be a single UOE of an application environment. In various embodiments, multiple transactions can be performed by the same thread, and each performance of the transaction by that thread is a separate unit of execution.

[0044] An application environment 32, 34 and 36 is typically monitored by an associated collector 62, 64 and 66, respectively. The collector 62, 64 and 66 gathers data about the units of execution 42 and 44, 46 and 48, 50 and 52, 54 and 56 of its associated application environment 32, 34, 36 and 38, respectively, and publishes data comprising communication event records as indicated by arrows 67, 68 and 69, to the managing server 20. The information gathered by the collector 62, 64 and 66 is used to provide a trace of the processing of a transaction within an application environment. Whenever a unit of execution inside a monitored application environment requests an external service from another unit of execution in a different application environment, the processing is said to leave the application environment.

[0045] In various embodiments, the processing of individual transactions is tracked across units of execution in different application environments 32, 34 and 36 using tokens. In some embodiments, a "composite" trace of the processing associated with a transaction is provided across the various application environments. For example, in various embodiments, using a graphical user interface, a user can view a method trace of a servlet within a J2EE container, see that the servlet made a request to a CICS region, switch to view a trace of the processing associated with that request in the CICS region where a CICS transaction or task is executed on its behalf, and follow a program trace of the CICS transaction. At this point the CICS transaction could send an MQ Series message to IMS through an MQ-IMS bridge, which invokes an IMS transaction which can be tracked in the same context as the servlet and the CICS transaction. In various embodiments, using the tokens, an end-to-end view of the processing of transactions across various application environments is provided with the ability to drill down to view the processing inside the various monitored application environments. Therefore, the processing related to a transaction can be viewed in the same context across different application environments in a computing network.

[0046] In some embodiments, the collector 62, 64 and 66, is implemented in an application management system (AMS) 72, 74 and 76, respectively, for example, WebSphere Studio Application Monitor (WSAM) or Tivoli® (Registered trademark of International Business Machines Corporation) Monitoring for Transaction Performance (TMTP). However, the application management system is not meant to be limited to WSAM or Tivoli TMTP, and other application management systems may be used.

[0047] In various embodiments, a UOE has a UOE identifier. In some embodiments, the UOE identifier is unique within an application environment. A thread is associated with a thread identifier; a TCA is associated with a TCA number; and a TCB is associated with a TCB number. In various embodiments, the UOE identifier is a thread identifier; a TCA number or a TCB number.

[0048] FIG. 2 depicts an embodiment of a UOE tag 82 which comprises a UOE identifier 84 and a timestamp (Timestamp) 86. FIG. 3A depicts an embodiment of a UOE tag 90 comprising a hosting thread identifier 92 as the UOE identifier, and a timestamp (Timestamp) 94. FIG. 3B depicts another embodiment of a UOE tag 96 which comprises a TCA number 98 as the UOE identifier, and a timestamp 100. In another embodiment, a TCB number is used as the UOE identifier instead of a TCA number. In some embodiments thread identifiers, TCA numbers and TCB numbers are reusable; therefore, a UOE identifier of an application environment may not be unique for that application environment. However, the thread identifiers, TCA numbers and TCB numbers are unique within an application environment for a sufficient duration of time so that, when used as a UOE identifier, their associated threads, TCAs, and TCBs, respectively, can be correctly identified.

[0049] Referring back to FIG. 1, the managing server 20 communicates with and manages the collectors. The managing server 20 stores communication event records in addition to other information. The managing server 20, a global publishing server (GPS) 80 correlates the processing of transactions across application environments 32, 34 and 36, respectively, based on the communication event records. Correlating the processing of transactions across application environments refers to associating the UOEs with each other based on their communications, either through invocation (synchronous or asynchronous) or through messaging. To correlate the processing of transactions across UOEs in different application environments, a token is generated. The token is sent from the originating UOE to the receiving UOE with the communication. The token is also sent to the managing server as part of a communication event record. In some embodiments, each token which is generated by a collector is distinct. In other embodiments, the tokens are distinct for a sufficient period of time so that the managing server 20, in particular, the GPS 80, can correctly correlate the processing of the transaction across the UOEs in different application environments. In various embodiments, the collectors send a communication event record comprising the token, a UOE identifier and timestamp, or UOE tag, to the managing server to notify the managing server of the attempt to communicate.

[0050] In various embodiments, to track composite transactions, a token is generated and exchanged between the originating UOE and the receiving UOE. Depending on the nature of the interaction, tokens are either generated by the collector using fields which are managed by that collector or generated using information from fields which are not managed by the collector. In some embodiments, the term "native token" refers to a token which is generated by the collector using fields which are managed by that collector. In other embodiments, the term "guest token" refers to a token which is generated by the collector using information from fields which are not managed by that collector, for example, using information from the communications interface structure which is managed by other applications. In some embodiments, if there is no available field in the communications interface structure which can be used to host a native token, a guest token is derived from fields in the communications interface structure the combination of which is "reasonably" unique. In other words, the guest token is sufficiently unique so that the associated UOEs and application environments can be correctly correlated. In some embodiments, the guest token allows the associated
UOEs and application environments to be correctly correlated in excess of a desired probability.

[0051] FIG. 4 depicts an embodiment of a native token 102. The native token 102 has an application environment identifier 104 and a trip counter 106. In various embodiments, the native token 102 has a predetermined length. In some embodiments, the predetermined length is four bytes; however, the predetermined length is not meant to be limited to four bytes and other numbers of bytes may be used. In various embodiments, the native token 102 contains binary (non-character) data. The application environment identifier 104 uniquely identifies the application environment originating a request. In various embodiments, the application environment identifier for each application environment associated with a managing server is distinct. In some embodiments, the application environment identifier 104 comprises the first n most significant bits of the token 102. The number of bits n is a configurable value and depends on the site transaction volume and the number of collectors managed by the managing server. For example, if n is equal to ten, there can be up to 1,024 collectors; if n is equal to eleven, the number of collectors is doubled to 2,048. The remaining bits (32-n) are used as the trip counter 106. In response to any unit of execution in an application environment making a request to leave that application environment, such as an invoke, the trip counter 106 is incremented by one. For example, if a request is made by a thread to leave its region, the trip counter 106 is incremented by one. If n is equal to ten, the trip counter 106 has a maximum value of 4,194,304 before wraparound. The managing server maintains a list which associates the application environment identifiers with their respective application environment names, and in some embodiments, other associated information. In some embodiments, in which an application environment comprises multiple regions, and a plurality of collectors are associated with a plurality of regions, the collectors associated with those regions use a single application environment identifier, typically stored in shared memory, which is used to represent all the regions, and the collectors use a single trip counter, which is also stored in shared memory.

[0052] For example, in various embodiments, non-message queue based communication uses native tokens 102 for both synchronous and asynchronous communication.

[0053] FIG. 5 depicts an embodiment of a guest token 110. In various embodiments, the guest token 110 comprises a message identifier (MsgId) 112 and a correlation identifier (CorrelId) 114. In some embodiments, guest tokens 110 are used for messaging systems, that is, message queue, based interactions between application environments. In various embodiments, the messaging system is WebSphere MQ Series; however, in other embodiments, other messaging systems may be used. In some embodiments, a guest token 110 has the following properties: the guest token 110 is unique within the application environment monitored by the collector to avoid collisions within the application environment; the guest token 110 has a length that is greater than the predetermined length of the native token to avoid collisions with native tokens; the guest token 110 remains with the message between putting the message on a message queue and retrieving the message from the message queue; and the guest token 110 contains binary data not subject to changes or conversion. In some embodiments, the guest token 110 has a length that is less than the predetermined length of the native token.

[0054] In various embodiments, the message identifier 112 is a byte string that is used to distinguish one message from another message. Typically, each message has a different message identifier 112. In some embodiments, the message identifier 112 is generated by the queue manager of the messaging system; and in other embodiments, the message identifier 112 is generated by the application or system.

[0055] In some embodiments, a correlation identifier 114 is a byte string that an application or collector can use to relate one message to another message, or to relate the message to other work that the application is performing. The correlation identifier 114 is either not generated and a value of (24 x ‘00’) is used, or alternately, the correlation identifier 114 is generated by the queue manager, or in some embodiments, the application.

[0056] The messaging system sends the message identifier 112 and correlation identifier 114 with the message. Therefore, in this embodiment, the collector does not modify any information in the message header. After executing instructions to place a message in the message queue such as MQPUT or MQPUT1, or instructions to retrieve a message such as MQGET, a message queue message descriptor, for example, an MQSeries message descriptor (MQMD), is available with the PUT-message options (MQPMO) or the GET-message options (MQGMO). In various embodiments, the message identifier 112 and correlation identifier 114 are available in the PUT-message options or the GET-message options which are associated with a message.

[0057] FIG. 6 depicts another embodiment of a guest token 120. In this embodiment, the guest token 120 comprises the correlation identifier 114 and message type 122. This embodiment of a guest token 120 is used when a communication is a response message from an MQ-IMS bridge. The message type 122 is typically set by the application and indicates whether the message is a reply to a prior message.

[0058] The following pseudo-code illustrates the generation of the guest tokens described in FIGS. 5 and 6 by a collector.

```
If (PutAppType=MQAT_IMS_BRIDGE or PutAppType=MQAT_XCF) and (MsgType = MQMT_REPLY)
    TOKEN = CorrelId || MsgType
    TOKEN LENGTH = MQ_CORREL_ID_LENGTH + 4
Else
    TOKEN = MsgId || CorrelId
    TOKEN LENGTH = MsgId_LENGTH + CorrelId_LENGTH
```

[0059] For example, when a J2EE application is initiating an IMS transaction through IMS Connect (via IMS Connect for Java (IC4J)), a native token is used. If this J2EE application was to use MQSeries through the MQ-IMS Bridge, a guest token is used. For a reply message, the guest token is derived from the CorrelId and MsgType fields in the MQSeries message descriptor and has a predetermined TOKEN LENGTH, which in some embodiments, is twenty-eight bytes. Otherwise, the guest token is derived from the
MsgID and Correlld fields in the MQSeries message descriptor and has a predetermined TOKEN LENGTH, which in some embodiments, is forty-eight bytes. However, the TOKEN LENGTH is not meant to be limited to twenty-eight or forty-eight bytes, and in other embodiments, other values for the TOKEN LENGTH may be used.

[0060] The collector communicates with the managing server according to a communication protocol comprising communication event records. The collector sends communication event records to the managing server. In various embodiments, a communication event record comprises an event type, a UOE identifier and a token, and in some embodiments, a timestamp. An originator communication event record is a communication event record which is sent by the originating UOE to the managing server. A receiver communication event record is a communication event record which is sent by the receiving UOE to the managing server. In various embodiments, the originator communication event record comprises an originator event type, an originating UOE identifier, a token, and, in some embodiments, a timestamp. Originator event types comprise GPS_INVOKE_START, GPS_PUT_START, GPS_INVOKE_END and GPS_PUT_END event records. A receiver communication event record comprises a receiver event type, a receiving UOE identifier, a token, and, in some embodiments, a timestamp. Receiver event types comprise GPS_RECEIVE_START, GPS_RECEIVE_END, GPS_GET_START and GPS_GET_END.

[0061] The event types may also be classified as start event types and end event types. Start event types comprise GPS_INVOKE_START, GPS_PUT_START, GPS_RECEIVE_START and GPS_GET_START. End event types comprise GPS_INVOKE_END, GPS_RECEIVE_END, GPS_PUT_END and GPS_GET_END.

[0062] In addition the event types may be classified as message-based and non-message based. The message-based event types indicate that a messaging system is used to send the communication. The message-based event types comprise GPS_PUT_START, GPS_PUT_END, GPS_GET_START and GPS_GET_END. The non-message-based event types indicate that a communication is not sent by a messaging system, for example, the communication may be sent through invocation (synchronous or asynchronous). The non-message-based event types comprise GPS_INVOKE_START, GPS_INVOKE_END, GPS_RECEIVE_START and GPS_RECEIVE_END.

[0063] In various embodiments, traps are configured to transfer control to the collector associated with an application environment in response to a unit of execution of that application environment attempting to communicate with another application environment, for example, by invocation or by sending a message. Traps are also configured to transfer control to the collector associated with an application environment in response to a unit of execution of that application environment receiving a communication from another application environment, for example, by returning from an invocation or by receiving a message. In some embodiments, traps are configured to transfer control to the collector associated with an application environment in response to returning from a call to a communication application programming interface (API). In addition, at initialization, in some embodiments, a trip counter in a collector is set to zero, or alternately, the trip counter in a collector is set equal to one. In various embodiments, the first, that is, initial, value of the token cannot be equal to zero.

[0064] In some embodiments, the collector associated with the application environment containing an originating UOE sends an originator communication event record comprising a GPS_INVOKE_START event type, the originating UOE identifier, a token and a timestamp, to the managing server in response to trapping a CICS or IMS request which is to be sent to a receiving UOE. The collector associated with the application environment containing the originating UOE sends a receiver communication event record comprising an event type of GPS_RECEIVE_START, a receiving UOE identifier, the received token and a timestamp, to the managing server in response to trapping the receipt of the request from the originating UOE. The receiving UOE processes the request. The request comprises the token. In various embodiments, in response to trapping the suspension or stopping of the processing at the receiving UOE to return control to the originating UOE, the collector associated with the application environment containing the receiving UOE sends a receiver communication event record comprising an event type of GPS_RECEIVE_END, the receiving UOE identifier, the token that was received in the request and a timestamp, to the managing server. The receiving UOE returns control to the originating UOE by sending a communication to the originating UOE with the token that was received in the request. In response to trapping the receipt of the communication from the receiving UOE which indicates that the execution of the request by the receiving UOE was stopped or suspended, the collector associated with the application environment containing the originating UOE sends an originator communication event record comprising an event type of GPS_INVOKE_END, the originating UOE identifier, the token that was passed in the communication from the receiving UOE and a timestamp, to the managing server, and control is returned to the originating UOE.

[0065] In other embodiments, the collector associated with the application environment containing the originating UOE sends an originator communication event record comprising an event type of GPS_PUT_START, the originating UOE identifier, a token and timestamp, to the managing server in response to sending a message using a message queue of a messaging system by, for example, issuing a PUT request or PUT API. The collector associated with the application environment containing the receiving UOE sends a communication event record comprising an event type of GPS_GET_START, the receiving UOE identifier, the received token and a timestamp, to the managing server in response to retrieving the message from the message queue. In various embodiments, a GET request or GET API is used to retrieve a message. In some embodiments, the receiver communication event record also indicates whether the GET request or API is destructive, for example, a “read” from a queue, or non-destructive, for example, a “browse” of the queue. Several units of execution in different regions can issue a browse against the same message in the queue, and those UOEIs will share the same token. The receiving UOE performs the request. The collector associated with the application environment containing the receiving UOE sends a receiver communication event record comprising an event type of GPS_GET_END, the receiving UOE identifier, the received token and a timestamp to the managing
server after regaining control from the GET request or API which retrieved the message from the message queue. Meanwhile, the collector associated with the application environment containing the originating UOE sends an originator communication event record comprising an event type of GPS_PUT_END, the originating UOE identifier, the token used in the communication event record having an event type of GPS_PUT_START and a timestamp, to the managing server after the UOE regains control from the PUT API or request to put a message on the message queue. Therefore, the originating UOE sends the originator communication event record having an event type of GPS_PUT_END asynchronously with respect to the receiving UOE.

[0066] FIG. 7 depicts a flowchart of an embodiment of the sending of an originator communication event record by a collector at an originating application environment in response to an attempt to communicate with a receiving UOE in a receiving application environment. A communication API is typically invoked to send the communication from the originating UOE to the receiving UOE. In step 132, in response to a UOE of the originating application environment attempting to communicate with a receiving UOE in a receiving application environment, the attempt to communicate is trapped and the collector associated with the originating application environment is activated or invoked. The receiving application environment is different from the originating application environment.

[0067] In step 134, the collector generates a token. In some embodiments, the token contains the application environment identifier of the originating application environment, and the value of the trip counter in the collector is incremented by one and stored in the token. In an alternate embodiment, the token is derived from environment variables, such as a message identifier and correlation identifier as described above. The token is hosted by, that is, contained in, a communication structure which depends on the nature of the communication. For example, in a request for a CICS service, the token is hosted, that is, stored, in the COMMAREA; in MQSeries, the token is part of the MQSeries message descriptor; and if IMS is the target of the request, the token is hosted by the Open Transaction Manager Access (OTMA) prefix user data.

[0068] In step 136, the collector sends an originator communication event record to the managing server. The originator communication event record has an event type of GPS_INVOKE_START or GPS_PUT_START, the UOE identifier of the originating UOE, the token and a timestamp. In some embodiments, the timestamp indicates when the request or message was sent by the UOE; in other embodiments, the timestamp indicates when the associated communication event record was sent to the managing server. In some embodiments, the UOE identifier and timestamp are contained in a UOE tag. In various embodiments, the token is a native token which contains the application environment identifier associated with the originating UOE and a trip counter as described above. In other embodiments, the token is a guest token. The collector sends an originator communication event record having an event type of GPS_INVOKE_START in response to a request for an IMS or CICS service and uses a native token. The collector sends an originator communication event record having an event type of GPS_PUT_START in response to a sending a message using the messaging system and uses a guest token. In some embodiments using a message queue, the originator communication event record also contains the queue name. In step 138, the collector exits.

[0069] In step 140, the communication is then sent to the receiving UOE with the token. The communication API which implements the communication passes the communication structure from the originating UOE to the receiving UOE. The communication structure contains the token as described above.

[0070] FIG. 8 depicts a flowchart of an embodiment of sending a receiver communication event record by a collector of a receiving UOE in a receiving application environment in response to receiving a communication from an originating UOE in an originating application environment. In step 142, the receipt of a communication comprising a token from an originating UOE in an originating application environment by a receiving UOE in a receiving application environment is trapped, and the collector associated with the receiving application environment is activated. The receiving application environment is different from the originating application environment.

[0071] In step 144, the collector extracts the token from the communication. In step 146, the collector acknowledges receipt of the communication by sending a receiver communication event record comprising an event type of GPS_RECEIVE_START or GPS_GET_START, the receiving UOE identifier, the received token and a timestamp, to the managing server. The timestamp indicates when the request or message was received, or alternately, when the receiver communication event record was sent to the managing server. In various embodiments, the UOE identifier and the timestamp are contained in a UOE tag. In some embodiments, in response to the request being an invocation of an IMS or CICS service, the event type is GPS_RECEIVE_START. In other embodiments, in response to the request being a message from a messaging system, the event type is GPS_GET_START. In some embodiments, the execution of the receiving UOE is initiated in response to the request or message; in other embodiments, the receiving UOE is already executing prior to receiving the request or message. In step 148, the collector exits. The unit of execution of the application environment processes the request or message.

[0072] FIG. 9 depicts a flowchart of an embodiment of sending a receiver communication event record by a collector of the receiving application environment in response to the receiving UOE attempting to send a communication which indicates that the receiving UOE is suspending or stopping execution. In step 152, an attempt to send a communication which indicates that the receiving UOE is suspending or stopping execution is trapped at the receiving application environment containing the receiving UOE, and the collector associated with the receiving application environment is activated. In step 154, the collector retrieves the token which was received in the communication from the originating UOE to provide a received token. In step 156, the collector stores the token in the communications structure associated with the communication which indicates that the receiving UOE is suspending or stopping execution. In step 158, the collector sends a receiver communication event record to the managing server. The receiver communication event record comprises an event type, the receiving UOE
identifier, the received token and a timestamp. The timestamp indicates when the receiver communication event record was sent. In various embodiments, the event type is GPS_RECEIVE_END. In step 160, the collector exits. In step 162, the communication which indicates that the receiving UOE is suspending or stopping execution is sent to the originating UOE with the token.

[0073] FIG. 10 depicts a flowchart of an embodiment of sending an originator communication event record by a collector at an originating application environment in response to receiving a communication that indicates that a receiving UOE is suspending or stopping execution. In step 164, at the originating UOE of an originating application environment, the receipt of a communication that execution of a receiving UOE has been suspended or stopped, is trapped. The communication comprises a token. The collector associated with originating application environment is activated. In step 166, the collector associated with the originating application environment extracts the token from the communication. In step 168, the collector associated with the originating application environment sends an originator communication event record to the managing server. The originator communication event record comprises a GPS_INVOKE_END event type, the originating UOE identifier, the token and a timestamp indicating when the originator communication event was sent. In step 170, the collector exits.

[0074] FIG. 11 depicts a flowchart of an embodiment of sending an originator communication event record by a collector associated with an originating application environment in response to the originating UOE receiving control back from sending a communication asynchronously. In step 172, the return from an API to send a communication to the receiving UOE using a messaging system is trapped at the originating application environment, and a collector associated with the originating application environment is activated. In some embodiments, the API is a PUT API to place a message in a message queue. In step 174, the collector associated with the originating application environment extracts the token from the communication. In step 176, the collector associated with the originating application environment sends an originator communication event record to the managing server comprising an event type, the originating UOE identifier, the token and a timestamp. In step 178, the collector exits. In some embodiments, the event type is GPS_PUT_END.

[0075] In an alternate embodiment, asynchronous invocation is used rather than a messaging system. In this embodiment, in step 172, the return from an API to send a communication to the receiving UOE using asynchronous invocation is trapped; and in step 176, the event type is GPS_INVOKE_END.

[0076] FIG. 12 depicts a flowchart of an embodiment of the sending of a receiver communication event record by a collector associated with a receiving application environment in response to a UOE receiving control back from an API to receive an asynchronous communication. In step 180, the return from the API to receive the communication is trapped at the receiving application environment, and the collector associated with the receiving application environment is activated. In some embodiments, the return is from a GET API to receive a communication from a messaging system. In other embodiments, the return is from an API which receives a communication from an asynchronous invocation. In step 182, the collector extracts the token from the communication. In step 184, the collector sends a receiver communication event record to the managing server. The receiver communication event record comprises an event type, the receiving UOE identifier, the token and a timestamp. In some embodiments, the event type is GPS_GET_END if a messaging system is used; and in other embodiments, the event type is GPS_RECEIVE_END if asynchronous invocation is used. In step 186, the collector exits.

[0077] In some embodiments, steps 134-138 of FIG. 7, steps 144-148 of FIG. 8, steps 154-160 of FIG. 9, steps 166-170 of FIG. 10, steps 174-178 of FIG. 11 and steps 182-186 of FIG. 12 are implemented in a single collector.

[0078] FIG. 13 depicts a flowchart of an embodiment of the managing server. In step 192, the managing server receives and stores originator and receiver communication event records. Each communication event record has a token.

[0079] In step 194, in the managing server, the GPS correlates UOEs across application environments based on the originator and receiver communication event records. The GPS determines whether two UOEs are correlated by matching the tokens of the receiver and originator communication event records. For example, a UOE associated with the UOE identifier of an originator communication event record which has an event type of GPS_INVOKE_START is correlated with a UOE associated with the UOE identifier of a receiver communication event record which has an event type of GPS_RECEIVE_START if the tokens of the originator and receiver communication event records are the same. A UOE associated with the UOE identifier of an originator communication event record which has an event type of GPS_INVOKE_END is correlated with a UOE associated with the UOE identifier of a receiver communication event record which has an event type of GPS_RECEIVE_END if the tokens of the originator and receiver communication event records are the same. A UOE associated with the UOE identifier of an originator communication event record which has an event type of GPS_PUT_START is correlated with a UOE associated with the UOE identifier of a receiver communication event record which has an event type of GPS_GET_START if the tokens of the originator and receiver communication event records are the same. A UOE associated with the UOE identifier of an originator communication event record which has an event type of GPS_PUT_END is correlated with a UOE associated with the UOE identifier of a receiver communication event record which has an event type of GPS_GET_END if the tokens of the originator and receiver communication event records are the same.

[0080] The GPS identifies the originating and receiving UOEs based on the UOE identifier sent in the originator and receiver communication event records, respectively. The managing server receives other protocol messages from the data collector which are used to establish the identity of a thread, TCA or TCB within an application environment. When a collector starts, the collector identifies itself uniquely to the managing server, including the type of the collector, that is, the type of application environment that the
collector is associated with, such as, J2EE, CICS or IMS. When the managing server receives a communication event record, the managing server identifies which collector is sending the communication event record, and therefore which application environment is associated with the communication event record. Therefore, based on which collector sent the communication event record and the UOE identifier of a communication event record, the managing server can identify the UOE and the application environment associated with the communication event record. In various embodiments, the managing server maintains information such as the names of the UOEs, their associated UOE identifiers, the name of their associated application environments and their associated application environments identifier. The GPS also determines what type of connection is used (synchronous/asynchronous invocation or messaging) by the event type of the communication event record. The GPS determines the order of events based on the timestamp. In some embodiments, the GPS determines whether a communication is synchronous or asynchronous based on the timestamp.

[0081] In step 196, the managing server provides a visual indication of the correlation between the originator and receiver communication event records, and therefore indicates the correlation between UOEs in different application environments.

[0082] Various examples of the communication protocol between the collectors and the managing server will now be described. The communication protocol works with synchronous and asynchronous communication. One difference between synchronous and asynchronous communication may be indicated by the difference in the timestamp. For example, for asynchronous communication the timestamp will be such that in some instances the communication event record having an event type of GPS_INVoke_END from the originating UOE will have a timestamp with a value which is prior to the value of the timestamp of the communication event record having an event type of GPS_RECEIVE_END from the receiving UOE. In other words, the originating UOE does not “block” while the receiving UOE is executing. In synchronous invocation, the originating UOE waits until the receiving UOE finishes executing the invocation, whereas in asynchronous invocation, the originating UOE does not wait and can perform other activities, such as invoking another UOE. Therefore, in some instances, the GPS can determine whether a communication is synchronous or asynchronous based on the timestamp.

[0083] FIG. 14 depicts a sequence diagram 200 illustrating an embodiment of tracking the processing of a transaction across application environments using synchronous communication. The vertical axis 202 represents time. FIG. 14 has three units of execution, UOE1, UOE2 and UOE3, 204, 206 and 208, respectively, and each unit of execution is in a different application environment. FIG. 14 also indicates when the UOEs 204, 206 and 208 start and stop executing with a “Start” label 212 and a “Stop” label 214. Table 1 illustrates the communication event records of FIG. 14 which are sent to and stored by the managing server. In the sequence diagram 200 of FIG. 14 and in Table 1, the communication event records are associated with an event record number (EVR) 210 having a value from one through sixteen which correspond to the order in which the event records are issued. In FIG. 14, the EVRs 210 are enclosed in a circle. In synchronous communication, the originating UOEs suspend execution until control is returned to them. In FIG. 14, a dashed line 216 indicates that the execution of a UOE is suspended. Table 1 below depicts various exemplary communication event records, including the event type, an exemplary UOE identifier with the TCA or TCB number, timestamp, and an exemplary token with the application environment identifier (AE id) and trip counter. In the example below, the timestamp contains an illustrative value as “hh:mm” where “hh” represents the hour using a twenty-four hour clock and “mm” represents the minute.

### TABLE 1

<table>
<thead>
<tr>
<th>EVR</th>
<th>Event Type</th>
<th>TCB number</th>
<th>Time stamp</th>
<th>AE id</th>
<th>Trip counter</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>GPS_INVoke_START</td>
<td>1</td>
<td>1300</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>GPS_RECEive_END</td>
<td>2</td>
<td>1302</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>GPS_INVoke_START</td>
<td>2</td>
<td>1304</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>GPS_RECEive_END</td>
<td>3</td>
<td>1305</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>GPS_RECEive_END</td>
<td>3</td>
<td>1308</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>GPS_INVoke_END</td>
<td>2</td>
<td>1309</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>GPS_INVoke_START</td>
<td>2</td>
<td>1310</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>GPS_RECEive_END</td>
<td>3</td>
<td>1311</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>9</td>
<td>GPS_RECEive_END</td>
<td>3</td>
<td>1312</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>10</td>
<td>GPS_INVoke_END</td>
<td>2</td>
<td>1313</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>11</td>
<td>GPS_INVoke_START</td>
<td>2</td>
<td>1314</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>12</td>
<td>GPS_RECEive_END</td>
<td>3</td>
<td>1315</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>13</td>
<td>GPS_RECEive_END</td>
<td>3</td>
<td>1316</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>14</td>
<td>GPS_INVoke_END</td>
<td>2</td>
<td>1317</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>15</td>
<td>GPS_RECEive_END</td>
<td>2</td>
<td>1318</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>16</td>
<td>GPS_INVoke_END</td>
<td>1</td>
<td>1319</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

[0084] For example, a first unit of execution UOE1 204 of a first application environment attempts to communicate with a second unit of execution UOE2 206 of a second application environment. In response, as indicated by event record number one 216, the collector associated with the first application environment UOE1 204 sends a communication event record having a GPS INVoke_START event type to the managing server. The execution of the first unit of execution UOE1 204 is suspended as indicated by dashed line 216. In response to receiving the communication from the first unit of execution UOE1 204, as indicated by event record number two, the collector associated with the second unit of execution UOE2 206 of the second application environment sends a communication event record having an event type of GPS RECEive_START to the managing server. The second unit of execution UOE2 206 attempts to communicate with the third unit of execution UOE3 208. In response, as indicated by event record number three, the collector associated with the second unit of execution UOE2
sends a communication event record having an event type of GPS_INVOKE_START to the managing server. The execution of the second unit of execution UOE2 206 is suspended. In response to receiving the communication event record from the second unit of execution UOE2 206, as indicated by event record number four, the collector associated with the third unit of execution UOE3 208 sends a communication event record having an event type of GPS_RECEIVE-_START to the managing server. In response to the third unit of execution UOE3 208 suspending execution, the collector associated with the third unit of execution UOE3 208 sends a communication event record having an event type of GPS_RECEIVE-_END to the managing server as indicated by event record number five. Control is transferred back to the second unit of execution UOE2 206. In response to control being transferred back to the second unit of execution UOE2 206, the collector associated with the second unit of execution UOE2 206 sends a communication event record having an event type of GPS_INVOKE_END event record to the managing server as indicated by event record number six. As shown by event record numbers seven to sixteen, the processing continues until control returns to the first unit of execution UOE1 204.

FIG. 15 depicts a sequence diagram 220 illustrating an embodiment of tracking the processing of a transaction across application environments using non-message driven asynchronous and synchronous communication. FIG. 15 illustrates three units of execution, UOE1, UOE2 and UOE3, 222, 224 and 226, respectively, and each unit of execution is in a different application environment. In FIG. 15, the start and stop time of the UOE's 222, 224 and 226, is also illustrated. In the sequence diagram and in Table 2, the communication event records below are numbered with an event record number (EVR) from one through sixteen, however, the event record numbers of FIG. 15 and Table 2 do not reflect the order in which the communication event records were issued. The event record numbers are used to associate the communication event records of Table 2 with FIG. 15. In FIG. 15, UOE1 222 communicates using synchronous invocation and does not suspend execution after invoking UOE2 224, as illustrated by the solid line 225 which indicates that UOE1 222 continues processing after the communication event record associated with event record number sixteen 228 is sent to the managing server. UOE2 224 and UOE3 226 communicate using synchronous invocation.

TABLE 2-continued

<table>
<thead>
<tr>
<th>EVR</th>
<th>Event Type</th>
<th>TCB number</th>
<th>Time stamp</th>
<th>AE id</th>
<th>Trip counter</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>GPS_INVOKE_END</td>
<td>2</td>
<td>1309</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>GPS_INVOKE_START</td>
<td>2</td>
<td>1310</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>GPS_RECEIVE_START</td>
<td>3</td>
<td>1311</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>9</td>
<td>GPS_RECEIVE_END</td>
<td>3</td>
<td>1312</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>10</td>
<td>GPS_INVOKE_END</td>
<td>2</td>
<td>1313</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>11</td>
<td>GPS_INVOKE_START</td>
<td>2</td>
<td>1314</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>12</td>
<td>GPS_RECEIVE_START</td>
<td>3</td>
<td>1315</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>13</td>
<td>GPS_PRECEIVE_END</td>
<td>3</td>
<td>1316</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>14</td>
<td>GPS_INVOKE_END</td>
<td>2</td>
<td>1317</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>15</td>
<td>GPS_RECEIVE_END</td>
<td>2</td>
<td>1318</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>16</td>
<td>GPS_INVOKE_END</td>
<td>1</td>
<td>1303</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

In FIG. 15, UOE1 222 is operating asynchronously with respect to UOE2 224 and UOE3 226. The communication event record having an event type of GPS_INVOKE-_END (labeled as event record number sixteen) 228 corresponding to the communication event record having an event type of GPS_INVOKE_START (labeled as event record number one) of UOE1 222 has a timestamp prior to the timestamp of the communication event records having event record numbers of 3 to 15 which indicates that UOE1 222 is not suspended while UOE2 224 and UOE3 226 are executing. Therefore UOE1 222 operates asynchronously, and the communication between UOE1 222 and UOE2 224 is asynchronous. UOE2 224 and UOE3 226 operate synchronously as described above with reference to FIG. 14.

FIG. 16 depicts a sequence diagram 230 illustrating an embodiment of tracking the processing of a transaction across application environments using message driven asynchronous communication. In FIG. 16, UOE1 and UOE2, 232 and 234, respectively, are executing asynchronously in different application environments and communicate with each other through messaging. The start and stop times of UOE1 232 and UOE2 234 are also illustrated in FIG. 16. The dashed lines of FIG. 16 indicate a time delay between sending and receiving a communication between UOE1 232 and UOE2 234. Table 3 depicts communication event records of FIG. 16 which were received and stored by the managing server. In FIG. 15, event record numbers (EVRs) are enclosed in a circle. The event record numbers of Table 3 are used to indicate corresponding communication event records in FIG. 16. The event record numbers do not correspond to the order in which the communication event records are sent to the managing server. In this example, the token is a guest token comprising a message identifier and a correlation identifier.
<table>
<thead>
<tr>
<th>EVR</th>
<th>Event Type</th>
<th>Hosting thread identifier</th>
<th>Time stamp</th>
<th>Message identifier</th>
<th>Correlation identifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>GPS_PUT_START</td>
<td>M3</td>
<td>1300</td>
<td>1</td>
<td>C1</td>
</tr>
<tr>
<td>2</td>
<td>GPS_GET_START</td>
<td>M3</td>
<td>1307</td>
<td>1</td>
<td>C1</td>
</tr>
<tr>
<td>3</td>
<td>GPS_GET_END</td>
<td>M3</td>
<td>1308</td>
<td>1</td>
<td>C1</td>
</tr>
<tr>
<td>4</td>
<td>GPS_PUT_END</td>
<td>M3</td>
<td>1301</td>
<td>1</td>
<td>C1</td>
</tr>
<tr>
<td>5</td>
<td>GPS_PUT_START</td>
<td>M2</td>
<td>1310</td>
<td>2</td>
<td>C2</td>
</tr>
<tr>
<td>6</td>
<td>GPS_GET_START</td>
<td>M3</td>
<td>1316</td>
<td>2</td>
<td>C2</td>
</tr>
<tr>
<td>7</td>
<td>GPS_GET_END</td>
<td>M3</td>
<td>1317</td>
<td>2</td>
<td>C2</td>
</tr>
<tr>
<td>8</td>
<td>GPS_PUT_END</td>
<td>M3</td>
<td>1311</td>
<td>2</td>
<td>C2</td>
</tr>
<tr>
<td>9</td>
<td>GPS_PUT_END</td>
<td>M3</td>
<td>1322</td>
<td>3</td>
<td>C3</td>
</tr>
<tr>
<td>10</td>
<td>GPS_GET_END</td>
<td>M3</td>
<td>1329</td>
<td>3</td>
<td>C3</td>
</tr>
<tr>
<td>11</td>
<td>GPS_GET_END</td>
<td>M3</td>
<td>1330</td>
<td>3</td>
<td>C3</td>
</tr>
<tr>
<td>12</td>
<td>GPS_PUT_END</td>
<td>M3</td>
<td>1323</td>
<td>3</td>
<td>C3</td>
</tr>
</tbody>
</table>

In this example, the request message specified that a response message is to be returned to the J2EE process. Upon completion of the transaction associated with the request message, the IMS application sends a response message using a PUT API to the J2EE process using the MQ-IMS bridge. The PUT API is trapped and the collector associated with the IMS application is invoked. As illustrated in the pseudo-code above, the collector generates a token (CorelliD, Msgtype) to avoid ambiguity between the tokens of the request message. The collector associated with the IMS application sends an originator communication event record comprising an event type of GPS_PUT_END, the hosting thread identifier of the J2EE process, the token (MsgID, CorelliD) and timestamp to the managing server. The IMS application processes a transaction associated with the request message.

In some embodiments, messages are sent to IMS using an MQ-IMS bridge. For example, a J2EE process sends a request message with a token (MsgID, CorelliD) to IMS through the MQ-IMS bridge using a PUT API, and the collector associated with the J2EE process sends an originator communication event record comprising an event type of GPS_PUT_START, the hosting thread identifier of the J2EE process, the token (MsgID, CorelliD) and timestamp to the managing server. The request message is routed to IMS with the token (MsgID, CorelliD), and IMS invokes a GET API to retrieve the message. The collector associated with the IMS application sends a receiver communication event record comprising an event type of GPS_GET_START, the UOE identifier of the IMS application, the received token (MsgID, CorelliD) and timestamp to the managing server. Since the request message does not come from IMS, the IMS application considers the token for that request message to be (MsgID, CorelliD). Because the messaging is asynchronous, the collector associated with the J2EE process sends an originator communication event record comprising an event type of GPS_PUT_END, the hosting thread identifier of the J2EE process, the token (MsgID, CorelliD), and timestamp to the managing server in response to the PUT API returning control to the J2EE process. In addition, the collector associated with the IMS application sends a receiver communication event record comprising a GPS_GET_END event type, the IMS application identifier, the received token (MsgID, CorelliD) and a timestamp to the managing server upon returning control from the GET API to the IMS application. The IMS application processes a transaction associated with the request message.
is a second application environment. The originating arrow 270 indicates that the first UOE is correlated with the second UOE. The originating arrow points to the right and indicates that the communication event record associated with the communication has an event type of start and indicates that the communication is an invocation. In particular, the method named “itos/cics/ectesterei/CTGGTestereECTIServlet.processRequest” of the first unit of execution invoked the second unit of execution. The GPS generated the arrow 270 based on matching tokens in the originating communication event record having an event type of GPS_INVOKE_=START for the first unit of execution and a receiver communication event record having an event type of GPS_RECEIVE_=START. The GPS correlated the first unit of execution 254 named “CTGTestereECTIServlet” in a first application environment named “ADCDPL.M2L2.M2L2.servqua.76” 256 with the second UOE named “CIC23QA1.DPLC1C1A” in a second application environment named “ADCDPL.M2L2.M2L2.CIC23QA1.bcc” 272 based on the token as described above. The uniform resource indicator (URI) 275 within the arrow 254 provides information naming the second application environment, the application server, and the type of event, in this example, an invoke. Clicking on the URI 275 causes a trace of the invoked UOE outside the current application environment, that is, the second UOE, to be displayed.

In FIG. 17A, the elapsed times are equal to zero because the events occurred within one millisecond of each other and the timestamp granularity of the managing server is one millisecond. For example, the elapsed time between the servlet entry of the “/CTGTestereECTIServlet.doPost” method and the method entry of the “itos/cics/ectesterei/CTGTestereECTIServlet.doPost” method is less than a millisecond; therefore, the elapsed time associated with the “itos/cics/ectesterei/CTGTestereECTIServlet doPost” method is shown as zero. The CPU time represents the amount of time used by the CPU in processing at a depth before proceeding to another depth. For example, the amount of CPU time used by “CTGTestereECTIServlet” servlet at depth 0 prior to calling the “itos/cics/ectesterei/CTGTestereECTIServlet doPost” method is less than a millisecond; therefore, the CPU time associated with the “itos/cics/ectesterei/CTGTestereECTIServlet doPost” servlet at depth 0 is shown as zero.

The term “queue names match” 273 is part of the page caption and indicates whether the names of the put and get queues match if box 274 is checked. If the names of the put and get queues do not match, then the box 276 associated with the “queue does not match” caption 277 is checked.

FIG. 17B illustrates a graphical user interface 280 illustrating a trace 282 of the processing performed in the CICS region in response to the invocation from the first UOE of FIG. 17A. A receiving arrow 284 contains a URI which, when clicked on, causes the trace of FIG. 17A to be displayed. The receiving arrow points to the left and indicates that the communication event record associated with the receiving UOE has an event type of start. By clicking on the URI in an originating or receiving arrow, a user can graphically follow the trace of various units of execution across different application environments. In FIG. 17B, the trace 282 is of the second UOE, named “CIC23QA1.DPLC1C1A” 286, in the second application environment, named “ADCDPL.M2L2.M2L2.CIC23QA1.bcc”. An originating arrow 290 indicates that the second UOE named “CIC23QA1.DPLC1C1A” in the “ADCDPL.M2L2.M2L2.CIC23QA1.bcc” application environment invokes a third UOE named “CIC23QA2.DPLC1C2A” in a third application environment, which is a CICS region named “ADCDPL.M2L2.M2L2.CIC23QA2.bcc”.

FIG. 17C is a graphical user interface 300 which depicts a trace 302 of the processing performed in the third application environment, that is, another CICS region, in response to the invocation from the second UOE of FIG. 17B. The receiving arrow 304 contains a URI which, when clicked on, causes the trace of the second UOE of FIG. 17B to be displayed. The third UOE 306 is named “CIC23QA2.DPLC1C2A” and is in a third application environment, that is, a CICS region, named “ADCDPL.M2L2.M2L2.CIC23QA2.bcc” 308. In this example, the processing “hangs,” that is, fails, at event “Program Exit” 310; therefore, no additional arrow is displayed. In this way, a user can follow the processing of a transaction throughout units of execution in different application environments and, in this example, determine where the processing failed.

FIG. 17 illustrates synchronous communication. However, if, in FIG. 17A, an asynchronous call was made from the servlet, that is, the first unit of execution in FIG. 17A, the first originating arrow 270 (FIG. 17A) is displayed, and a receiving arrow back to the first unit of execution under the first originating arrow 270, with the same contents as the originating arrow, is also displayed to indicate the return from the PUT API. The receiving arrow is displayed even though the third unit of execution is in a “hang” state, because the first unit of execution is not waiting for the second unit of execution or the third unit of execution to complete.

In some embodiments, in response to a token of a receiver communication event record of a receiving UOE which has an event type of GPS_RECEIVE END matching a token of an originating communication event record of an originating UOE which has an event type of GPS_INVOKE END, a receiving arrow is displayed in a graphical user interface which displays a trace of processing of the receiving UOE, and an originating arrow is displayed in the graphical user interface which displays a trace of the processing of the originating UOE.

FIG. 18 depicts an embodiment of an illustrative computer system 360 which uses various embodiments of the present invention. The computer system 360 comprises a processor 362, display 364, input interfaces (UI) 366, communications interface 368, memory 370 and output interface(s) 372, all conventionally coupled by one or more busses 374. The input interfaces 366 comprise a keyboard 376 and a mouse 378. The output interface 372 comprises a printer 380. The communications interface 368 is a network interface (NI) that allows the computer 360 to communicate via the network 382. The communications interface 368 may be coupled to the network 382 via a transmission medium 364 such as a network transmission line, for example twisted pair, coaxial cable or fiber optic cable. In another embodiment, the communications interface 368 provides a wireless
interface, that is, the communications interface 368 uses a wireless transmission medium.

0099 The memory 370 generally comprises different modalities, illustratively semiconductor memory, such as random access memory (RAM), and disk drives. In various embodiments, the memory 370 stores an operating system 388 and the managing server 20 which comprises the GPS 80, an application environment 400 and associated collector 402.

0100 Computer systems 390 and 392 are coupled to the network 382 via transmission mediums 394 and 396, respectively. Each computer system 390 and 392 comprises an application environment 32 and 36 and an associated collector 62 and 66, respectively. The application environments 32 and 36 and their associated collectors 62 and 66 are stored in a memory on computer systems 390 and 392, respectively.

0101 In various embodiments, the specific software instructions, data structures and data that implement various embodiments of the present invention are typically incorporated in the managing server 20 and collectors 62, 66 and 402. Generally, an embodiment of the present invention is tangibly embodied in a computer-readable medium, for example, the memory 370, and is comprised of instructions which, when executed by the processor 362, cause the computer system 360 to utilize the present invention. The memory 370 may store the software instructions, data structures and data for any of the operating systems 388, managing server 20, application environment 400 and collector 402 in semiconductor memory, in disk memory, or a combination thereof. Other computer memory devices presently known or that become known in the future, or combination thereof, may be used for memory 370.

0102 The operating system 388 may be implemented by any conventional operating system such as the AIX® (Registered Trademark of International Business Machines Corporation), UNIX®, Windows® (Registered Trademark of Microsoft Corporation), Linux® (Registered trademark of Linus Torvalds), Solaris® (Registered trademark of Sun Microsystems Inc.), HP-UX® (Registered trademark of Hewlett-Packard Development Company, L.P.), z/OS and MVS operating systems.

0103 In various embodiments, the present invention may be implemented as a method, computer system, apparatus or article of manufacture using standard programming and/or engineering techniques to produce software, firmware, hardware, or any combination thereof. The term “article of manufacture” (or alternatively, “computer program product”) as used herein is intended to encompass a computer program accessible from any computer-readable device, carrier or media. In addition, the software in which various embodiments are implemented may be accessible through the transmission medium, for example, from a server over the network. The article of manufacture in which the code is implemented also encompasses transmission media, such as the network transmission line and wireless transmission media. Thus the article of manufacture also comprises the medium in which the code is embedded. Those skilled in the art will recognize that many modifications may be made to this configuration without departing from the scope of the present invention.

0104 The exemplary computer system illustrated in FIG. 18 is not intended to limit the present invention. Other alternative hardware environments may be used without departing from the scope of the present invention.

0105 The foregoing detailed description of various embodiments of the invention has been presented for the purposes of illustration and description. It is not intended to be exhaustive or to limit the invention to the precise form disclosed. Many modifications and variations are possible in light of the above teachings. It is intended that the scope of the invention be limited not by this detailed description, but rather by the claims appended thereto.

What is claimed is:

1. A computer-implemented method of tracking the processing of a transaction, comprising:

receiving, by a managing server, a first communication event record from a first collector associated with a first application environment having a first unit of execution, said first communication event record comprising a first unit of execution identifier and a first token;

receiving, by said managing server, a second communication event record from a second collector associated with a second application environment having a second unit of execution, said second communication event record comprising a second unit of execution identifier and a second token; and

determining, by said managing server, whether said first unit of execution and said second unit of execution are associated with processing said transaction based on said first token and said second token.

2. The method of claim 1 wherein said determining determines that said first unit of execution and said second unit of execution are associated with said processing said transaction in response to said first token matching said second token.

3. The method of claim 2 further comprising:

displaying, by said managing server, an indicator that indicates that said first unit of execution and said second unit of execution are associated with said processing said transaction.

4. The method of claim 1 further comprising:

in response to an attempt to send a communication by said first unit of execution to said second unit of execution, generating, by a first collector, said first token;

sending, by said first collector, said first communication event record comprising said first token and said first unit of execution identifier to said managing server;

sending said communication from said first unit of execution to said second unit of execution;

in response to receiving said communication by said second unit of execution, storing, by a second collector, a value of said first token in said second token; and

sending, by said second collector, said second communication event record comprising said second token and said second unit of execution identifier to said managing server.

5. The method of claim 1 wherein at least one of said first token and said second token is a native token.

6. The method of claim 5 wherein said native token comprises an application environment identifier and a trip counter, said method further comprising:
incrementing said trip counter in response to an attempt to send a communication by said first unit of execution to said second unit of execution.
7. The method of claim 1 wherein at least one of said first token and said second token is a guest token.
8. The method of claim 7 wherein said guest token comprises a message identifier and a correlation identifier.
9. The method of claim 7 wherein said guest token comprises a message type and a correlation identifier.
10. The method of claim 1 further comprising:
in response to an attempt to send a communication by said first unit of execution to said second unit of execution, generating, by a first collector, said first token;
sending, by said first collector, said first communication event record comprising said first token and said first unit of execution identifier, and also comprising a start event type, to said managing server;
sending, asynchronously, said communication from said first unit of execution to said second unit of execution; and
in response to said sending said communication, sending by said first collector, a third communication event record comprising said first unit of execution identifier and said first token, and also comprising an end event type, to said managing server.
11. The method of claim 1 wherein said first communication event record comprises a first timestamp, and said second communication event record comprises a second timestamp;
determining an elapsed time based on said first and second timestamps.
12. The method of claim 1 wherein said first and second communication event records further comprise an event type indicating that a communication associated with said first and second communication event records, respectively, is message-based.
13. The method of claim 1 wherein said first and second communication event records further comprise an event type indicating that a communication associated with said first and second communication event records, respectively, is non-message-based.
14. The method of claim 1 wherein said first application environment is associated with a Java process, and said second application environment is associated with a legacy software system.
15. An article of manufacture comprising a computer usable medium embodying one or more instructions executable by a computer for performing a method of tracking the processing of a transaction, said method comprising:
receiving, by a managing server, a first communication event record from a first collector associated with a first application environment having a first unit of execution, said first communication event record comprising a first unit of execution identifier and a first token;
receiving, by said managing server, a second communication event record from a second collector associated with a second application environment having a second unit of execution, said second communication event record comprising a second unit of execution identifier and a second token; and
receiving, by said managing server, a second communication record comprising said first token and said second unit of execution identifier, and also comprising a start event type, to said managing server; and
receiving, by said managing server, a second communication record comprising said first token and said second unit of execution identifier, and also comprising a start event type, to said managing server; and
determining, by said managing server, whether said first unit of execution and said second unit of execution are associated with processing said transaction based on said first token and said second token.
16. The article of manufacture of claim 15 wherein said first token and said second token are a native token.
17. The article of manufacture of claim 16 wherein said first token and said second token is a native token.
18. The article of manufacture of claim 15 wherein said managing server is a Java process, and said second application environment is associated with a legacy software system.
19. An article of manufacture comprising a computer usable medium embodying one or more instructions executable by a computer for performing a method of tracking the processing of a transaction, said method comprising:
receiving, by a managing server, a first communication event record from a first collector associated with a first application environment having a first unit of execution, said first communication event record comprising a first unit of execution identifier and a first token;
receiving, by said managing server, a second communication event record from a second collector associated with a second application environment having a second unit of execution, said second communication event record comprising a second unit of execution identifier and a second token; and
receiving, by said managing server, a second communication record comprising said first token and said second unit of execution identifier, and also comprising a start event type, to said managing server; and
receiving, by said managing server, a second communication record comprising said first token and said second unit of execution identifier, and also comprising a start event type, to said managing server; and
determining, by said managing server, whether said first unit of execution and said second unit of execution are associated with processing said transaction based on said first token and said second token.
16. The article of manufacture of claim 15 wherein said first token and said second token are a native token.
17. The article of manufacture of claim 16 wherein said first token and said second token is a native token.
18. The article of manufacture of claim 15 wherein said managing server is a Java process, and said second application environment is associated with a legacy software system.
19. An article of manufacture comprising a computer usable medium embodying one or more instructions executable by a computer for performing a method of tracking the processing of a transaction, said method comprising:
receiving, by a managing server, a first communication event record from a first collector associated with a first application environment having a first unit of execution, said first communication event record comprising a first unit of execution identifier and a first token;
receiving, by said managing server, a second communication event record from a second collector associated with a second application environment having a second unit of execution, said second communication event record comprising a second unit of execution identifier and a second token; and
receiving, by said managing server, a second communication record comprising said first token and said second unit of execution identifier, and also comprising a start event type, to said managing server; and
receiving, by said managing server, a second communication record comprising said first token and said second unit of execution identifier, and also comprising a start event type, to said managing server; and
determining, by said managing server, whether said first unit of execution and said second unit of execution are associated with processing said transaction based on said first token and said second token.
16. The article of manufacture of claim 15 wherein said first token and said second token are a native token.
17. The article of manufacture of claim 16 wherein said first token and said second token is a native token.
18. The article of manufacture of claim 15 wherein said managing server is a Java process, and said second application environment is associated with a legacy software system.
in response to sending said asynchronous communication, sending by said first collector, a third communication event record comprising said first unit of execution identifier and said first token, and also comprising an end event type, to said managing server.

25. The article of manufacture of claim 15 wherein said first communication event record comprises a first timestamp, and said second communication event record comprises a second timestamp; wherein said method further comprises:

determining an elapsed time based on said first and second timestamps.

26. The article of manufacture of claim 15 wherein said first and second communication event records further comprise an event type indicating that a communication associated with said first and second communication event records, respectively, is message-based.

27. The article of manufacture of claim 15 wherein said first and second communication event records further comprise an event type indicating that a communication associated with said first and second communication event records, respectively, is non-message-based.

28. A system to track the processing of a transaction, comprising:

a plurality of communication event records comprising originator communication event records and receiver communication event records, each communication event record being associated with a unit of execution of one of a plurality of application environments, each communication event record having a token, one or more of said originator communication event records being start event records; and

a managing server to determine whether one unit of execution in one of said application environments and another unit of execution in another one of said application environments are associated with processing said transaction based on said token of said originator communication event records and said receiver communication event records.

29. The computer system of claim 28 wherein said managing server determines that said one unit of execution and said another unit of execution are associated with said processing said transaction in response to a token of said originator communication event records matching a token of said receiver communication event records.

30. The computer system of claim 29 wherein said managing server also displays an indicator that indicates that said one unit of execution and said another unit of execution are associated with said processing said transaction.