US 20060248386A1

(19) United States

(12) Patent Application Publication (10) Pub. No.: US 2006/0248386 A1

Barga et al. (43) Pub. Date: Nov. 2, 2006

(54) PERSISTENT STATEFUL COMPONENT-BASED APPLICATIONS VIA AUTOMATIC RECOVERY

(75) Inventors: **Roger S. Barga**, Newcastle, WA (US); **David B. Lomet**, Redmond, WA (US)

Correspondence Address:
WOODCOCK WASHBURN LLP
(MICROSOFT CORPORATION)
ONE LIBERTY PLACE - 46TH FLOOR
PHILADELPHIA, PA 19103 (US)

(73) Assignee: **Microsoft Corporation**, Redmond, WA

(21) Appl. No.: **11/399,759**

(22) Filed: **Apr. 7, 2006**

Related U.S. Application Data

(63) Continuation of application No. 10/879,324, filed on Jun. 29, 2004, now Pat. No. 7,093,162, which is a continuation of application No. 09/946,092, filed on Sep. 4, 2001, now Pat. No. 6,820,218.

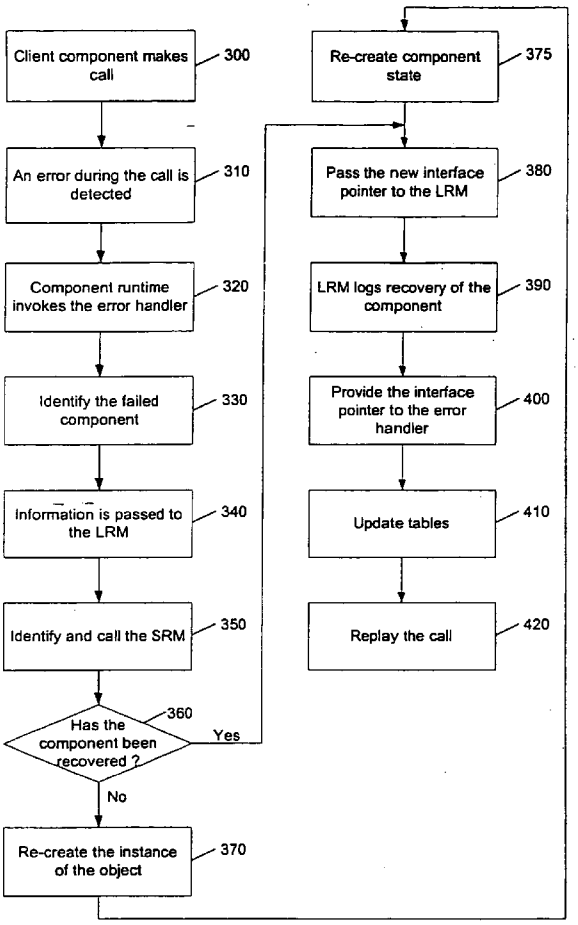
Publication Classification

(51) **Int. Cl.**
G06F 11/00 (2006.01)

(52) **U.S. Cl.** **714/15**

(57) **ABSTRACT**

Persistent components are provided across both process and server failures, without the application programmer needing take actions for component recoverability. Application interactions with a stateful component are transparently intercepted and stably logged to persistent storage. A "virtual" component isolates an application from component failures, permitting the mapping of a component to an arbitrary "physical" component. Component failures are detected and masked from the application. A virtual component is remapped to a new physical component, and the operations required to recreate a component and reinstall state up to the point of the last logged interaction is replayed from the log automatically.



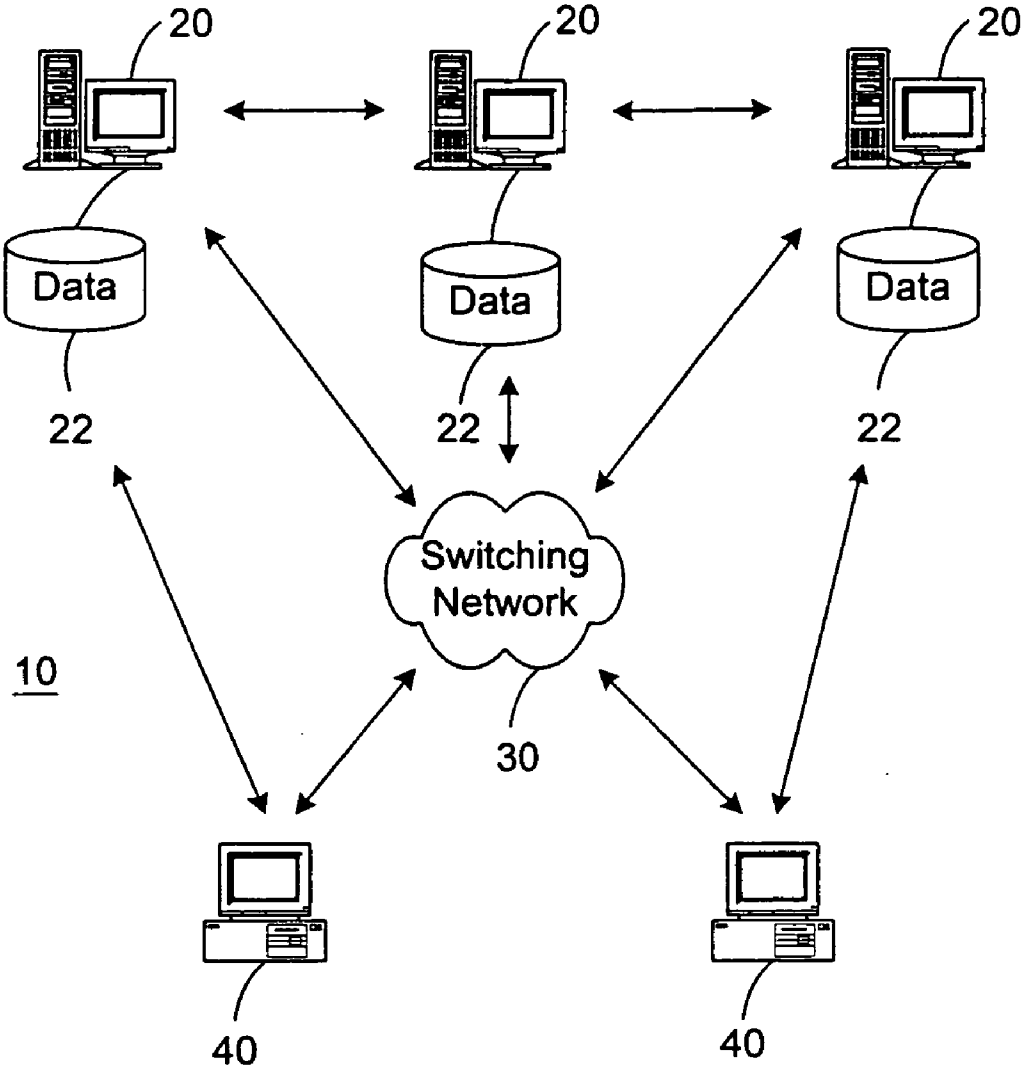


Fig. 1

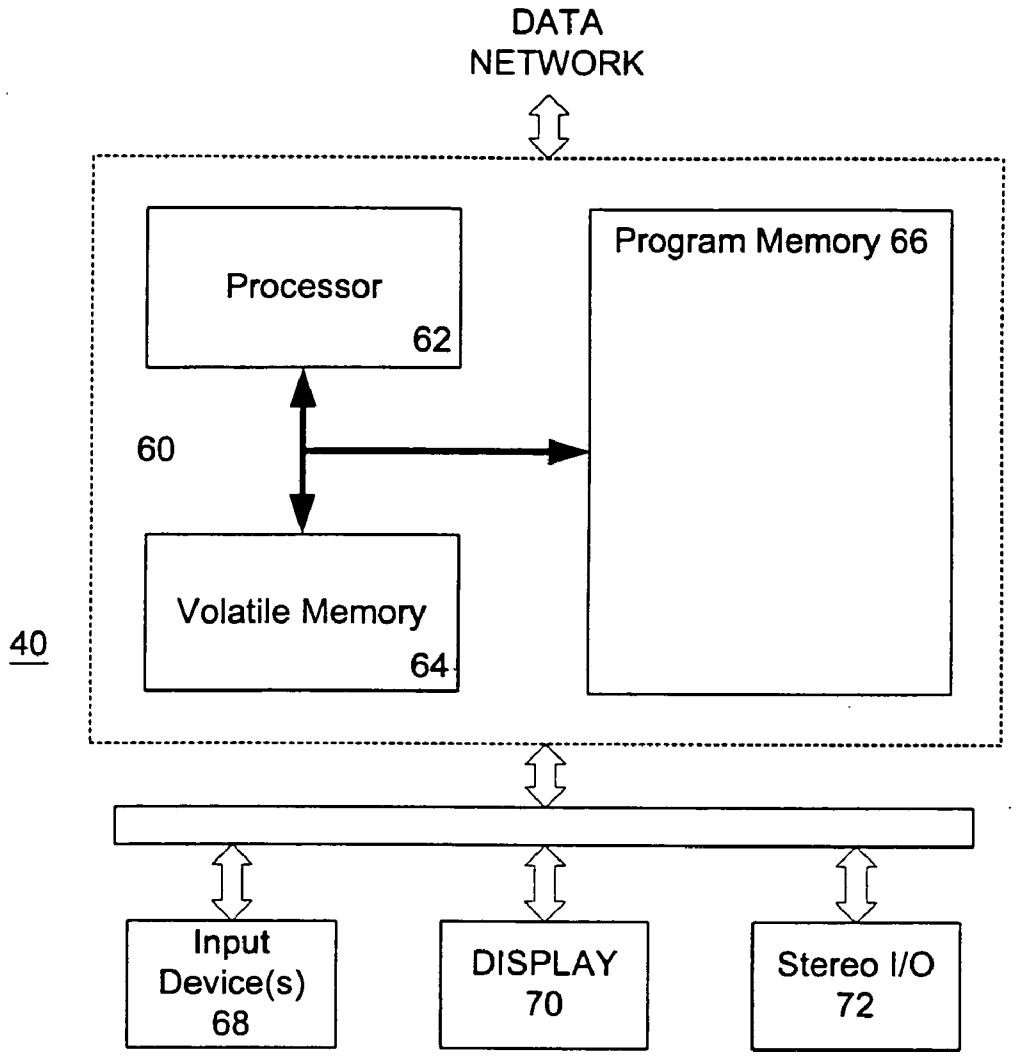

DATA
NETWORK
Processor
62
Program Memory 66
60
Volatile Memory
64
40
Input
Device(s)
68
DISPLAY
70
Stereo I/O
72

Fig. 2

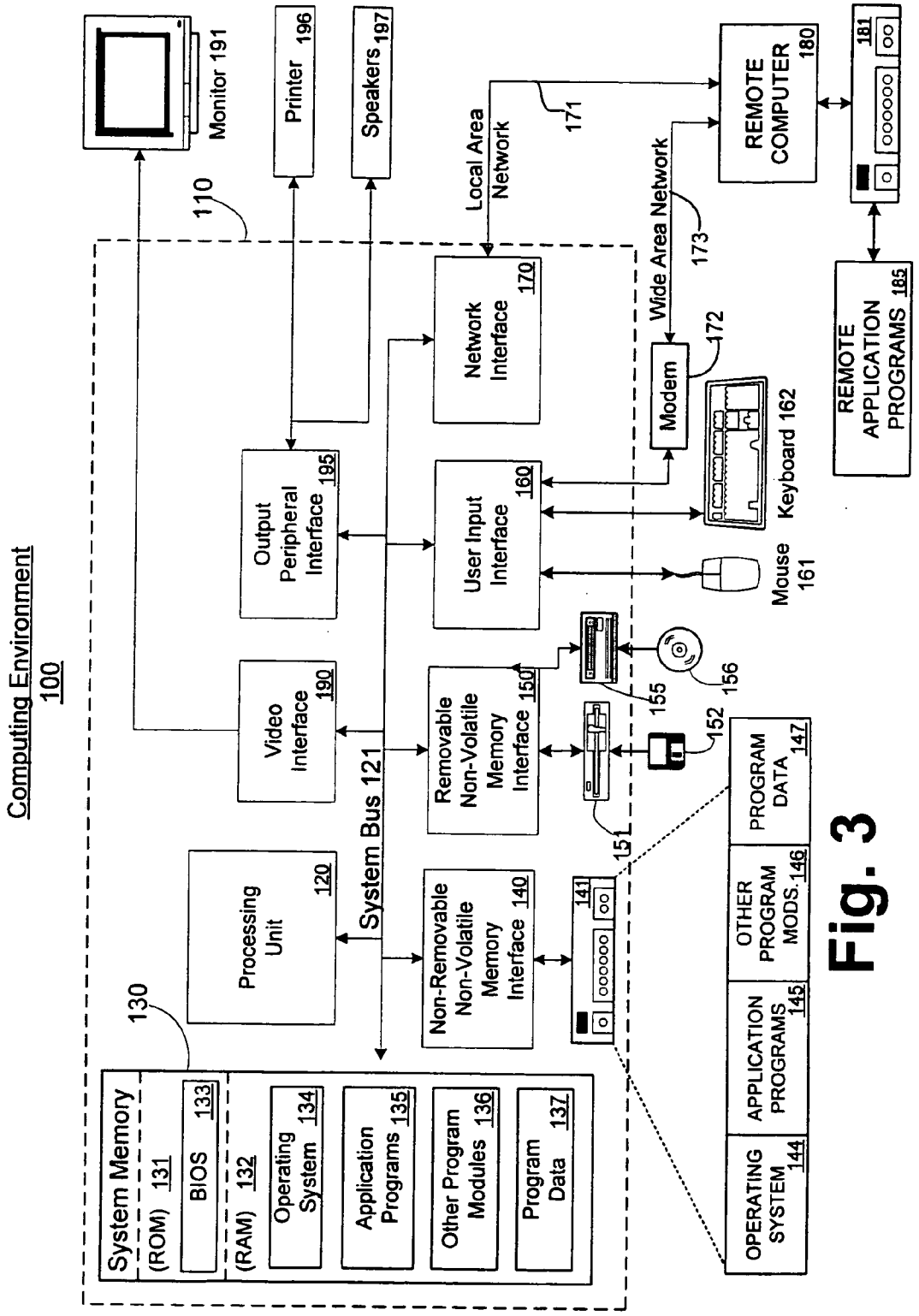

Computing Environment
100
110
Monitor 191
Printer 196
Speakers 197
Local Area Network
171
REMOTE COMPUTER 180
181
Wide Area Network
173
Network Interface 170
Modem
172
REMOTE APPLICATION PROGRAMS 185
Output Peripheral Interface 195
User Input Interface 160
Keyboard 162
Mouse 161
Video Interface 190
Removable Non-Volatile Memory Interface 150
155
156
152
151
System Bus 121
Processing Unit 120
Non-Removable Non-Volatile Memory Interface 140
141
130
System Memory
(ROM) 131
BIOS 133
(RAM) 132
Operating System 134
Application Programs 135
Other Program Modules 136
Program Data 137
OPERATING SYSTEM 144
APPLICATION PROGRAMS 145
OTHER PROGRAM MODS. 146
PROGRAM DATA 147

Fig. 3

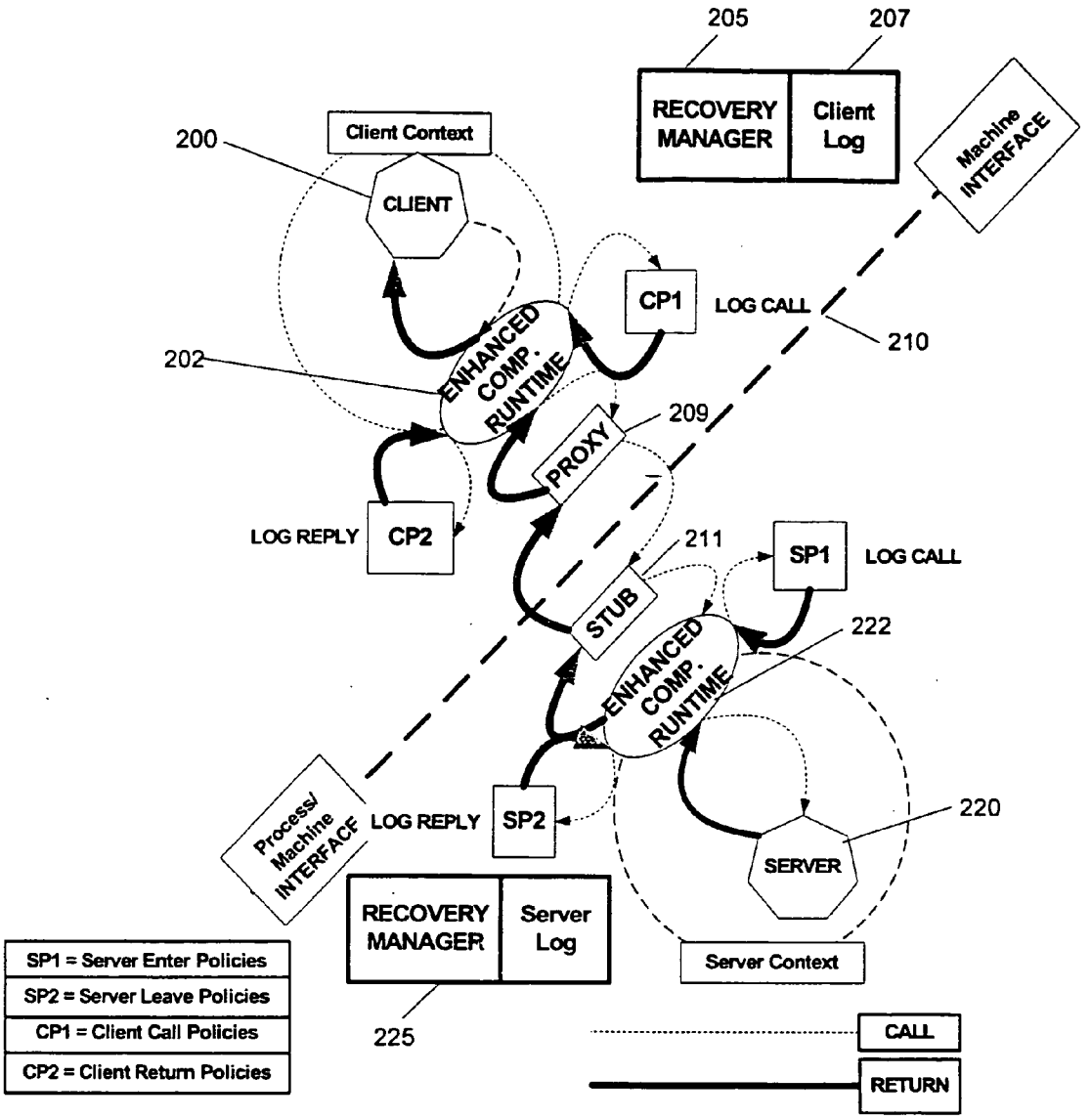

205
207
RECOVERY MANAGER
Client Log
Machine INTERFACE
200
Client Context
CLIENT
CP1
LOG CALL
210
202
ENHANCED COMP. RUNTIME
209
PROXY
LOG REPLY
CP2
211
SP1
LOG CALL
STUB
222
ENHANCED COMP. RUNTIME
Process/ Machine INTERFACE
LOG REPLY
SP2
220
SERVER
RECOVERY MANAGER
Server Log
Server Context
225
SP1 = Server Enter Policies
SP2 = Server Leave Policies
CP1 = Client Call Policies
CP2 = Client Return Policies
CALL
RETURN

Fig. 4

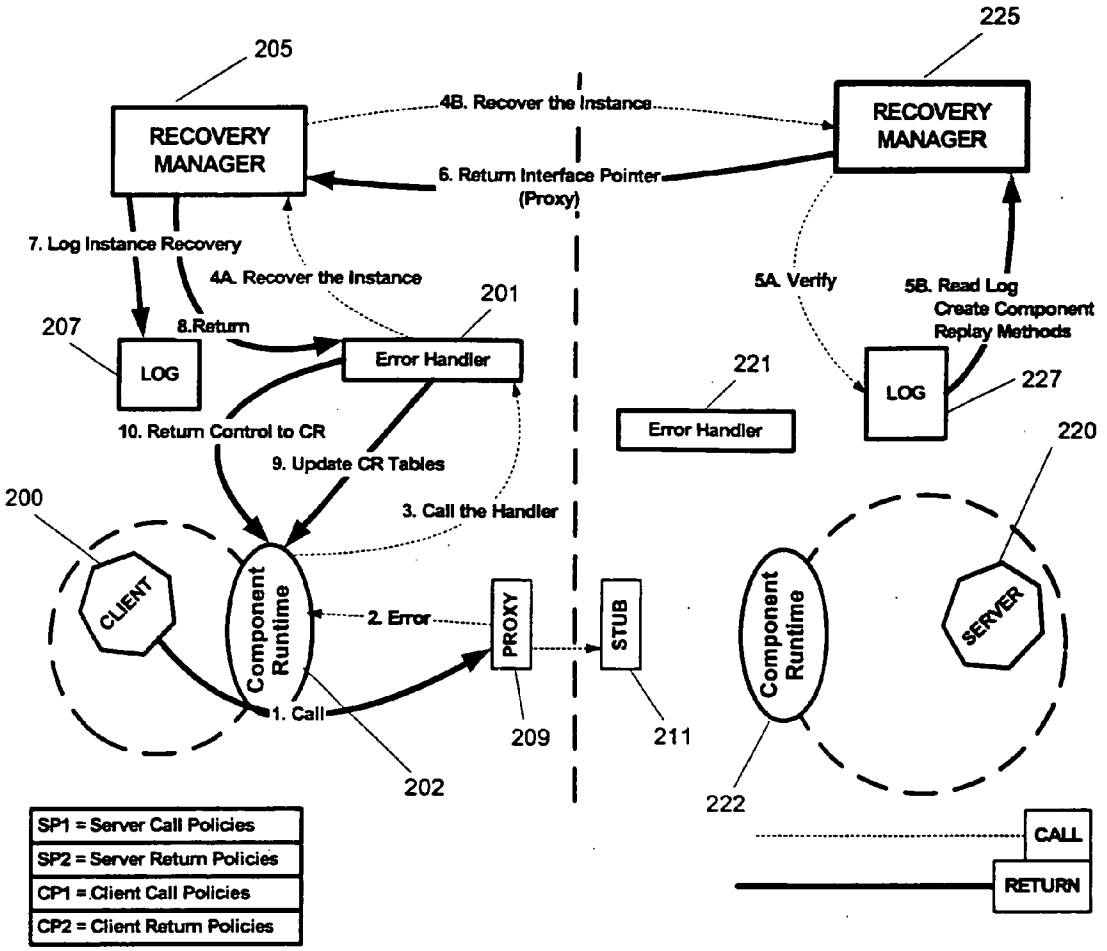

205
225
RECOVERY MANAGER
RECOVERY MANAGER
4B. Recover the Instance
6. Return Interface Pointer (Proxy)
7. Log Instance Recovery
4A. Recover the Instance
201
207
8.Return
LOG
Error Handler
10. Return Control to CR
9. Update CR Tables
3. Call the Handler
200
CLIENT
Component Runtime
2. Error
PROXY
STUB
1. Call
209
211
202
5A. Verify
5B. Read Log
Create Component
Replay Methods
221
Error Handler
LOG
227
220
Component Runtime
SERVER
222
SP1 = Server Call Policies
SP2 = Server Return Policies
CP1 = Client Call Policies
CP2 = Client Return Policies
CALL
RETURN

Fig. 5

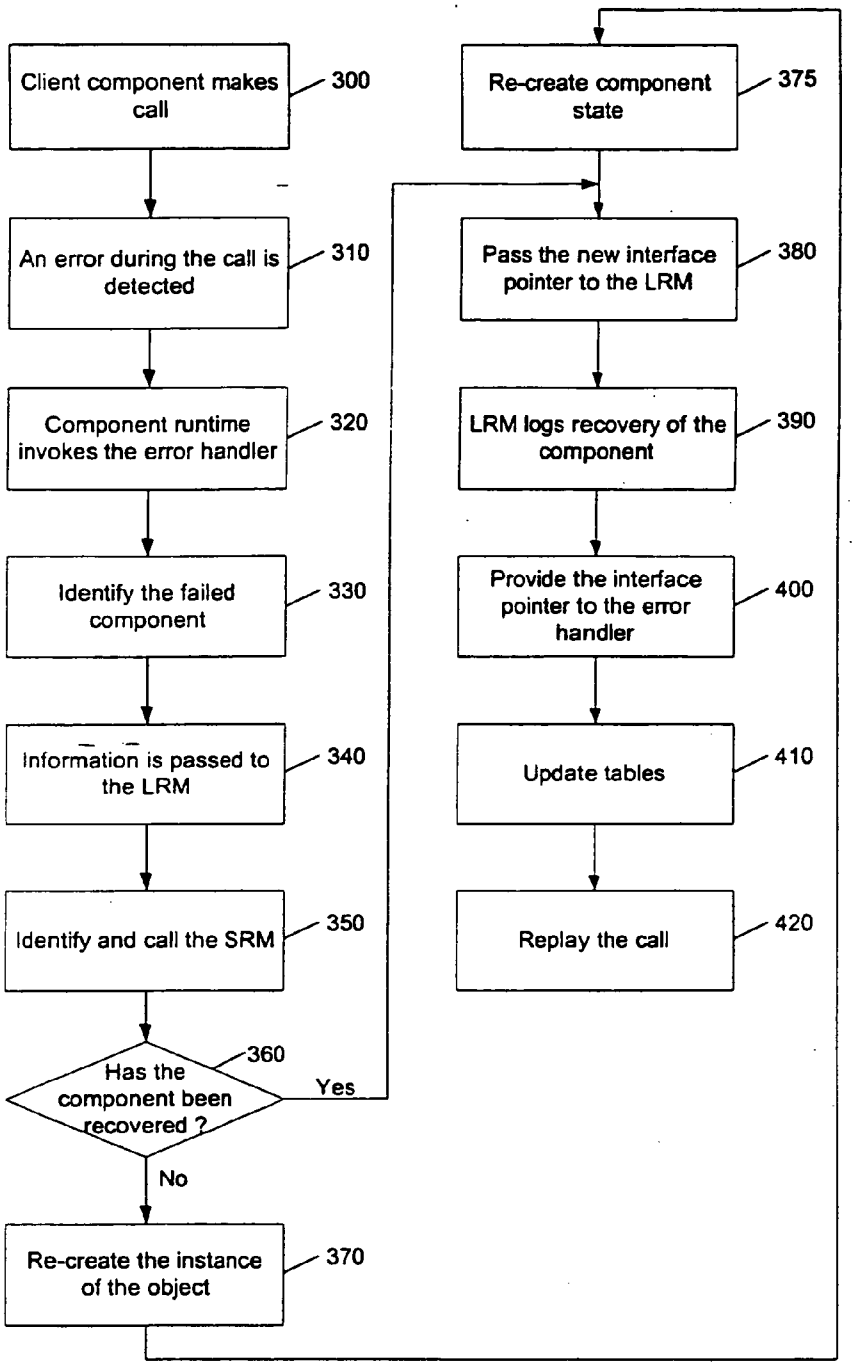

Client component makes call
300
An error during the call is detected
310
Component runtime invokes the error handler
320
Identify the failed component
330
Information is passed to the LRM
340
Identify and call the SRM
350
Has the component been recovered ?
360
Yes
No
Re-create the instance of the object
370
Re-create component state
375
Pass the new interface pointer to the LRM
380
LRM logs recovery of the component
390
Provide the interface pointer to the error handler
400
Update tables
410
Replay the call
420

Fig. 6

Type	Comp. CLSID	Comp. Location Info	Comp. Context	Iface ID	Logical ID	

TYPE = Create
Component CLSID = identify the class
Component location information = identity of the machine
Component context = context in which the component is to be created
I-face ID = identify the interface currently being used
Component Logical ID = logical identifier assigned to the component

Fig. 7

Type	Client CLSID	Client log. inst ID	Server CLSID	Server log. inst ID	I-face ID	Method Index	Method log. ID	Args, Retval

TYPE = (Call | Return)(First|Replay)
Client CLSID = identify the class
Client log. instance ID = identify the instance
Server CLSID = identify the class
Server log. instance ID = identify the instance
I-face ID = identify the interface currently being used
Method index = method called
Method log. ID = instance of method call

Fig. 8

Type	Component CLSID	Component log. inst ID	LastFlushedMethod

TYPE = LOG-FLUSH|CHECKPOINT
Component CLSID = identify the class
Component log. instance ID = identify the instance
LastFlushedMethod = corresponds to the method log. ID

Fig. 9

PERSISTENT STATEFUL COMPONENT-BASED APPLICATIONS VIA AUTOMATIC RECOVERY

CROSS-REFERENCE TO RELATED APPLICATIONS

[0001] This application is a continuation of U.S. patent application Ser. No. 10/879,324, filed on Jun. 29, 2004, which is a continuation of U.S. patent application No. 09/946,092, filed on Sep. 4, 2001, now issued as U.S. Pat. No. 6,820,218.

FIELD OF THE INVENTION

[0002] The present invention relates generally to the field of computer systems and applications that execute on them and, more particularly, to the automatic recovery of stateful application components when the computer system or process supporting the components fails.

BACKGROUND OF THE INVENTION

[0003] A component is a discrete unit of code that delivers a specified set of services through specified interfaces. Components provide the services that clients request at run time. A stateful component maintains private state resulting from the execution of one or more method calls. Thus, for example, a client application connecting to a component can have a "conversation" with the component that spans multiple method calls. The component can retain state across those method calls.

[0004] Building stateful components is a useful approach in application design. Programmers typically write stateful applications that retain state across component method calls. This provides the information necessary for correct and successful execution across transaction boundaries. However, such stateful components have availability and scalability limitations. One problem with stateful applications is the risk of losing the volatile state as a result of component failure when the system or process supporting the stateful application fails. Such a failure typically requires human intervention to repair or restart the application. Service outages can be very long because of this. A conventional response to this problem is to insist that applications be stateless. A stateless application has "no meaningful state maintained between transactions". The loss of the component state prevents the masking of system failures from clients, and degrades application availability.

[0005] In view of the foregoing, there is a need for systems and methods that overcome the limitations and drawbacks of the prior art.

SUMMARY OF THE INVENTION

[0006] The present invention provides persistent stateful components via transparent logging and automatic recovery. Persistent component state is based on method logging, in which interactions between components are intercepted and stably logged. Information sufficient to re-create a component and reinstall state up to the point of the last logged interaction (e.g., via redo recovery of the logged method calls) is persisted on stable storage. In the event of a system failure, aspects of the present invention automatically recreate failed components and recover lost state, without the stateful application itself needing to take measures for its recovery.

[0007] An embodiment of the present invention allows stateful application components to begin and end transactions, and, because these components have a state outside of transactions (i.e., a state that can persist across system failures), they are able to recognize transaction error codes, and act on them accordingly. For example, such an application can test the error code and decide whether to simply re-execute the transaction or change input parameters before re-execution, and decide at what point to abandon the effort and return an error code to the caller describing what has happened.

[0008] Thus, component-based applications can be written "naturally" as stateful programs. The present invention deals with system failures by logging component interactions and possibly checkpointing state to ensure that the application state can be automatically recovered should a failure occur.

[0009] Other features of the invention are described below.

BRIEF DESCRIPTION OF THE DRAWINGS

[0010] The foregoing summary, as well as the following detailed description of preferred embodiments, is better understood when read in conjunction with the appended drawings. For the purpose of illustrating the invention, there is shown in the drawings exemplary constructions of the invention; however, the invention is not limited to the specific methods and instrumentalities disclosed. In the drawings:

[0011] **FIG. 1** illustrates a high level example of a distributed computing environment in which the invention may be implemented;

[0012] **FIG. 2** shows an exemplary configuration of a client implemented as a computer;

[0013] **FIG. 3** is a block diagram showing an exemplary computing environment in which aspects of the invention may be implemented;

[0014] **FIG. 4** illustrates an exemplary position and relationship of various functional pieces of a component-based application running in accordance with the present invention;

[0015] **FIG. 5** illustrates an exemplary flow of control in server failure in accordance with the present invention;

[0016] **FIG. 6** is a flow chart of an exemplary recovery method in accordance with the present invention; and

[0017] **FIGS. 7, 8**, and **9** show various exemplary log record formats in accordance with the present invention.

DETAILED DESCRIPTION OF PREFERRED EMBODIMENTS

[0018] Overview

[0019] The present invention is directed to making normally volatile and stateful compo nents, e.g., COM components, persistent across both process and server failures, without the component itself needing to be aware of its recoverability. Persistent components provide improved application availability and reduce the application-programming task of coping with system failures. The present invention incorporates logging, in which interactions

between stateful volatile components are captured and stably logged. Using "virtual" components isolates the application from component failures, permitting the mapping of a virtual component to an arbitrary "physical" component. Component failures are detected and masked from the application program, and the virtual component is re-mapped to a new physical component into which the virtual component state is installed. Virtual components can also be re-mapped to physical components to enable scalability and load balancing. The result is persistent stateful components that can survive system failures without applications being aware of the outage.

[0020] The present invention enables robust applications by providing transparent recovery from various kinds of system failures. This enhances application availability by avoiding the extended down-time that such failures can produce. The present invention "wraps" component-based applications so as to perform logging that will enable transparent recovery from crashes. The ability to programmatically respond to transaction aborts is provided because application logic can exist outside of a transaction.

[0021] The present invention provides high availability by performing redo recovery for application components using its own logging and recovery infrastructure. Replay of an application component replaces those interactions with the logged effects of the interactions that took place originally. Hence, replay may be much faster than original execution.

[0022] Computing Environment

[0023] **FIG. 1** illustrates a high level example of a distributed computing environment **10** in which the invention may be implemented. A plurality of servers **20**, each having memory **22**, are interconnected, either directly or through an optional switching network **30**. A plurality of clients **40** are connected to the servers **20**, either directly or through the optional switching network **30**. Each of the clients **40** and servers **20** are described in further detail below.

[0024] **FIG. 2** shows an exemplary configuration of a client **40** implemented as a computer. It includes a central processing unit **60** having a processor **62**, volatile memory **64** (e.g., random access memory (RAM)), and program memory **66** (e.g., read only memory (ROM), flash, disk drive, floppy disk drive, CD-ROM, and the like). The client **40** has one or more input devices **68** (e.g., keyboard, mouse, etc.), a computer display **70** (e.g., VGA, SVGA), and a stereo I/O **72** for interfacing with a stereo system.

[0025] The client **40** runs an operating system that supports multiple applications. The operating system is preferably a multitasking operating system that allows simultaneous execution of multiple applications. The operating system employs a graphical user interface windowing environment that presents the applications or documents in specially delineated areas of the display screen called "windows." One preferred operating system is a Windows® brand operating system sold by Microsoft Corporation, such as Windows® 95 or Windows® NT or other derivative versions of Windows®. It is noted, however, that other operating systems that provide windowing environments may be employed, such as the Macintosh operating system from Apple Computer, Inc. and the OS/2 operating system from IBM.

[0026] **FIG. 3** illustrates a more detailed example of a suitable computing system environment **100** in which the invention may be implemented. Each server and client can incorporate the environment **100** of **FIG. 3**. The computing system environment **100** is only one example of a suitable computing environment and is not intended to suggest any limitation as to the scope of use or functionality of the invention. Neither should the computing environment **100** be interpreted as having any dependency or requirement relating to any one or combination of components illustrated in the exemplary operating environment **100**.

[0027] The invention is operational with numerous other general purpose or special purpose computing system environments or configurations. Examples of well known computing systems, environments, and/or configurations that may be suitable for use with the invention include, but are not limited to, personal computers, server computers, handheld or laptop devices, multiprocessor systems, microprocessor-based systems, set top boxes, programmable consumer electronics, network PCs, minicomputers, mainframe computers, distributed computing environments that include any of the above systems or devices, and the like.

[0028] The invention may be described in the general context of computer-executable instructions, such as program modules, being executed by a computer. Generally, program modules include routines, programs, objects, components, data structures, etc. that perform particular tasks or implement particular abstract data types. The invention may also be practiced in distributed computing environments where tasks are performed by remote processing devices that are linked through a communications network or other data transmission medium. In a distributed computing environment, program modules and other data may be located in both local and remote computer storage media including memory storage devices.

[0029] With reference to **FIG. 3**, an exemplary system for implementing the invention includes a general purpose computing device in the form of a computer **110**. Components of computer **110** may include, but are not limited to, a processing unit **120**, a system memory **130**, and a system bus **121** that couples various system components including the system memory to the processing unit **120**. The system bus **121** may be any of several types of bus structures including a memory bus or memory controller, a peripheral bus, and a local bus using any of a variety of bus architectures. By way of example, and not limitation, such architectures include Industry Standard Architecture (ISA) bus, Micro Channel Architecture (MCA) bus, Enhanced ISA (EISA) bus, Video Electronics Standards Association (VESA) local bus, and Peripheral Component Interconnect (PCI) bus (also known as Mezzanine bus).

[0030] Computer **110** typically includes a variety of computer readable media. Computer readable media can be any available media that can be accessed by computer **110** and includes both volatile and nonvolatile media, removable and non-removable media. By way of example, and not limitation, computer readable media may comprise computer storage media and communication media. Computer storage media includes both volatile and nonvolatile, removable and non-removable media implemented in any method or technology for storage of information such as computer readable instructions, data structures, program modules or other data. Computer storage media includes, but is not limited to, RAM, ROM, EEPROM, flash memory or other memory

technology, CD-ROM, digital versatile disks (DVD) or other optical disk storage, magnetic cassettes, magnetic tape, magnetic disk storage or other magnetic storage devices, or any other medium which can be used to store the desired information and which can accessed by computer **110**. Communication media typically embodies computer readable instructions, data structures, program modules or other data in a modulated data signal such as a carrier wave or other transport mechanism and includes any information delivery media. The term "modulated data signal" means a signal that has one or more of its characteristics set or changed in such a manner as to encode information in the signal. By way of example, and not limitation, communication media includes wired media such as a wired network or direct-wired connection, and wireless media such as acoustic, RF, infrared and other wireless media. Combinations of any of the above should also be included within the scope of computer readable media.

[**0031**] The system memory **130** includes computer storage media in the form of volatile and/or nonvolatile memory such as ROM **131** and RAM **132**. A basic input/output system **133** (BIOS), containing the basic routines that help to transfer information between elements within computer **110**, such as during start-up, is typically stored in ROM **131**. RAM **132** typically contains data and/or program modules that are immediately accessible to and/or presently being operated on by processing unit **120**. By way of example, and not limitation, **FIG. 3** illustrates operating system **134**, application programs **135**, other program modules **136**, and program data **137**.

[**0032**] The computer **110** may also include other removable/non-removable, volatile/nonvolatile computer storage media. By way of example only, **FIG. 3** illustrates a hard disk drive **141** that reads from or writes to non-removable, nonvolatile magnetic media, a magnetic disk drive **151** that reads from or writes to a removable, nonvolatile magnetic disk **152**, and an optical disk drive **155** that reads from or writes to a removable, nonvolatile optical disk **156**, such as a CD-ROM or other optical media. Other removable/non-removable, volatile/nonvolatile computer storage media that can be used in the exemplary operating environment include, but are not limited to, magnetic tape cassettes, flash memory cards, digital versatile disks, digital video tape, solid state RAM, solid state ROM, and the like. The hard disk drive **141** is typically connected to the system bus **121** through a non-removable memory interface such as interface **140**, and magnetic disk drive **151** and optical disk drive **155** are typically connected to the system bus **121** by a removable memory interface, such as interface **150**.

[**0033**] The drives and their associated computer storage media, discussed above and illustrated in **FIG. 3**, provide storage of computer readable instructions, data structures, program modules and other data for the computer **110**. In **FIG. 3**, for example, hard disk drive **141** is illustrated as storing operating system **144**, application programs **145**, other program modules **146**, and program data **147**. Note that these components can either be the same as or different from operating system **134**, application programs **135**, other program modules **136**, and program data **137**. Operating system **144**, application programs **145**, other program modules **146**, and program data **147** are given different numbers here to illustrate that, at a minimum, they are different copies. A user may enter commands and information into the computer **20** through input devices such as a keyboard **162** and pointing device **161**, commonly referred to as a mouse, trackball or touch pad. Other input devices (not shown) may include a microphone, joystick, game pad, satellite dish, scanner, or the like. These and other input devices are often connected to the processing unit **120** through a user input interface **160** that is coupled to the system bus, but may be connected by other interface and bus structures, such as a parallel port, game port or a universal serial bus (USB). A monitor **191** or other type of display device is also connected to the system bus **121** via an interface, such as a video interface **190**. In addition to the monitor, computers may also include other peripheral output devices such as speakers **197** and printer **196**, which may be connected through an output peripheral interface **195**.

[**0034**] The computer **110** may operate in a networked environment using logical connections to one or more remote computers, such as a remote computer **180**. The remote computer **180** may be a personal computer, a server, a router, a network PC, a peer device or other common network node, and typically includes many or all of the elements described above relative to the computer **110**, although only a memory storage device **181** has been illustrated in **FIG. 3**. The logical connections depicted in **FIG. 2** include a local area network (LAN) **171** and a wide area network (WAN) **173**, but may also include other networks. Such networking environments are commonplace in offices, enterprise-wide computer networks, intranets and the Internet.

[**0035**] When used in a LAN networking environment, the computer **110** is connected to the LAN **171** through a network interface or adapter **170**. When used in a WAN networking environment, the computer **110** typically includes a modem **172** or other means for establishing communications over the WAN **173**, such as the Internet. The modem **172**, which may be internal or external, may be connected to the system bus **121** via the user input interface **160**, or other appropriate mechanism. In a networked environment, program modules depicted relative to the computer **110**, or portions thereof, may be stored in the remote memory storage device. By way of example, and not limitation, **FIG. 3** illustrates remote application programs **185** as residing on memory device **181**. It will be appreciated that the network connections shown are exemplary and other means of establishing a communications link between the computers may be used.

[**0036**] Stateful Applications

[**0037**] Stateful application components can begin and end transactions, and, because these components have a state outside of transactions (and a state that persists across system failures), they can recognize transaction error codes, and act on them accordingly. For example, such an application can test the error code and decide whether to re-execute the transaction or change input parameters before re-execution, and can decide at what point to abandon the effort and itself return an error code describing what has happened.

[**0038**] Stateful application components can be recovered (have their state re-created) by logging non-deterministic events, and replaying those events during recovery. Between the logged events, the application itself can be re-executed because the non-deterministic events have been made deter-

ministic by logging and subsequently replaying the now deterministic events from the log. Such components are called piece-wise deterministic (PWD) components.

[0039] Replay after failure involves a different process and thread than used by the original execution. Therefore, objects are virtualized by providing logical identifiers for them that are independent of how the objects are mapped to processes or threads. A logical identifier identifies the class code and the persistent state of an instance. During execution, these persistent logical identifiers are mapped to the specific threads and processes realizing the objects.

[0040] Transparent persistence is achieved with the use of virtual components. Virtual components isolate an application from physical component failure. Virtual components are saved during processing, and after a failure, physical components are re-mapped using the saved virtual components.

[0041] Component Execution

[0042] In order to perform component recovery (via replay), the present invention logs, during normal execution, information sufficient to re-create a component and recover its state to the last logged interaction. An activation hook (interception code invoked during component creation) is implemented that logs information related to component creation, and policy functions (interception code invoked during method calls and returns) are implemented that log interactions (calls and responses) between client and server components. Each component is associated with a context that defines the boundary at which interception takes place.

[0043] **FIG. 4** illustrates the position and relationship of various functional elements of a component-based application running in accordance with the present invention. The elements **200** and **220** can be any component, such as that acting as a client, server, or both, and that the invention should not be limited to the recovery of components acting in the role of a server responding to requests from other client components. For simplicity, it is assumed that the client **200** and server **220** are disposed in different processes or machines, so there are two different component runtimes (a client-side component runtime **202** and server-side component runtime **222**), along with a proxy **209** and a stub **211**.

[0044] With respect to **FIG. 4**, there is a logging policy on each of CP**1**, CP**2**, SP**1**, SP**2**, and there is an activation hook registered with both client (through a policy) and server contexts. The activation hook records (logs) information regarding the creation of an object. An error handler (not shown in **FIG. 4**; elements **201**, **221** in **FIG. 5**) is registered with each component runtime **202**, **222**. Furthermore, two recovery managers **205**, **225** are provided because of the machine boundary **210** between the server **220** and the client **200**.

[0045] When the client **200** wants to call the server **220**, the call is intercepted by the component runtime **202** (also referred to as an enhanced component runtime) which then invokes caller policies CP**1**, CP**2**. One of the caller policies is logging which captures the message in a log **207** and the call gets passed to the proxy, a unit of code in the local process that stands in for the server. The proxy **209** then sends the message, e.g. a method call message, across the boundary **210** to the stub **211**. The stub acts for the client in the server process, The stub calls the server side component runtime **222**, which invokes policies SP**1**, SP**2** and then delivers the call.

[0046] Error Handling and Recovery

[0047] A recoverable failure may occur for several reasons including: (1) the proxy associated with the server object returns a failure indicating that it is disconnected from the server—this happens when the server process shuts down on a failure; (2) one of the policies in the server context nullifies the call with an interface specific error that indicates a recoverable failure; and (3) the server object itself returns the above error on a call. In such cases, the component runtime on the client side will detect an error. Note that in the first case the server process does not exist any more. When the client's component runtime gets an error on a call, it calls an error handler that is already registered with the component runtime.

[0048] The component runtime passes the identity of the call that failed to the error handler. The information includes the identity of the server, the identity of the client, and the identity of the method. The error handler determines if the cause of the error is from component failure and, if so, invokes the local recovery manager (LRM) to recover the failed component. The error is masked from the client and recovery is initiated.

[0049] If recovery is successful, the recovery manager will return a new component reference to the recovered component to the error handler. The error handler updates component runtime tables on the client and logs the recovery event. Finally, the error handler returns control to the component runtime for normal method processing. Otherwise, if recovery is unsuccessful, the error handler unmasks the error by returning a code indicating failure to the component runtime. Moreover, the error handler allows for canceling or repeating the call, and can examine the call state (e.g., ID, method, parameters, etc.).

[0050] **FIG. 5** illustrates an exemplary flow of control in server failure in accordance with the present invention. Regarding server failure, it is assumed a client **200** detects the failure during processing of a call, or more accurately the component runtime **202** on a client **200** detects the error. The recovery manager that recovers the failed component can be on the same machine as the failed component, though this is not necessary

[0051] In **FIG. 5**, the server recovery manager (SRM) **225** takes the following steps to recover the failed component. It is first verified that the component has not already been recovered. An instance of the failed component is re-created. Methods from the log are replayed against the re-created component and the component is brought to the state of the last logged interaction. Control is returned to the local recovery manager (LRM) **205**, with the interface pointer passed back to the recovered component. The error handler **201** on the client **200** updates the component runtime tables and returns control to the component runtime **202** so that method execution can resume normally.

[0052] Component Recovery

[0053] **FIG. 6** is a flow chart of an exemplary recovery method in accordance with the present invention. At step **300**, a call is made by the client component through the

component runtime. At step **310**, an error is detected during the call and processing is returned to the component runtime. At step **320**, the component runtime invokes the error handler, passing information related to the call. At step **330**, the error handler uses this information to identify the failed component and then gathers additional information that will be used to process the forthcoming recovery request. At step **340**, the error handler then calls the LRM and passes the information. The error handler blocks until this call returns.

[**0054**] At step **350**, the LRM identifies the SRM, which is responsible for the component, via a table lookup or other discovery process, and then calls the SRM using a timeout (in case the SRM process or machine is down), passing information about the failed component. To identify the recovery manager, given the server, the LRM performs a lookup in a global table, for example, to get a handle for the SRM. This lookup could also be implemented as a discovery service.

[**0055**] The SRM is called to recover the failed component. At step **360**, the SRM checks an in-memory structure or log to determine whether the component has already been recovered with the client having a stale interface pointer (reference). With a lazy update, for example, it is possible another client or process already recovered the component, and this was not broadcast. Duplicate call elimination provides exactly-once semantics. When components recover, they might resend a duplicate message. The system detects whether or not a message has already been received and eliminates duplicate messages, to guarantee that messages are executed exactly once. An exemplary method of exactly-once execution comprises uniquely identifying messages at a sender, detecting an original message and at least one duplicate message at a receiver, eliminating the at least one duplicate message, and providing a reply to the at least one duplicate message that is the same as a reply to the original message.

[**0056**] If the component has been recovered, processing continues at step **380**; otherwise, at step **370**, the SRM reads its log to locate the activation log record and then re-creates the instance of that object. This new instance will have a different interface pointer, but is assigned the same logical ID as the old one. For example, the SRM creates a special component on the server using the COM cocreate call. This object will be used to issue a "create instance" call. The SRM preferably assigns the component the same logical ID as before.

[**0057**] At step **375**, the SRM reads the log and replays methods on the server to re-create the component state. The SRM scans the log, using the logical ID of the failed component to identify log records pertaining to the failed component. The SRM creates the new component instance by reading the activation log record from the server log and replaying the creation call. During recovery the SRM intercepts method return values. Next, the SRM invokes these logged method calls one by one against the new component, dropping the method return values. When completed, handles and other references to the component are updated with the new interface pointer.

[**0058**] At step **380**, once the object has been brought back to the state it was in before failure, the SRM returns to the LRM, passing back the new interface pointer. The SRM returns to the LRM with information about the new server instance, so the component runtime tables on the client side can be modified. Also, the SRM logs the recovery of the component on the server log.

[**0059**] At step **390**, the LRM logs the recovery of the component, along with the new identifier, then at step **400**, returns the new interface pointer to the error handler. At step **410**, the error handler updates component runtime tables with the new interface pointer, and returns a status code to the component runtime indicating success. At step **420**, the component runtime replays the method call using the marshaled arguments (which are placed on the stack). If there is an error during the method call, processing will return to step **320**; otherwise, control returns to the client.

[**0060**] Logging

[**0061**] Regarding information that is logged to facilitate component recovery, it is specified below what is logged, where the value will originate (e.g., context variable, from the object create call or method call, off the stack, etc.), and the format of the resulting log record. Sources of information include, but are not limited to: information associated with object creation; the mapping between the logical identifier for a component and its current interface pointer; and information associated with method call and return, at both client and server. There is other desirable information for recovery that can be logged, such as component state checkpoints, session information between client and server components, recovery actions, etc.

[**0062**] During client creation, an activation hook intercepts the creation call for the component. The activation hook code contacts the recovery manager, providing arguments pertaining to the creation of the component. The recovery manager generates a new logical identifier for the component, and then logs the creation information. An exemplary log record format is shown in **FIG. 7**. A logical identifier is associated with the machine on which the component is created, location information of the context, and the class ID (CLSID) for the component.

[**0063**] For server creation, the client calls for the creation of the server on the same or different machine. During component creation the activation hook code running on the server can gather the CLSID, location information, logical ID, etc. for the server component in a manner similar to client creation.

[**0064**] At this point, both client and server components have been created and log records detailing the creation information have been written to the log. To complete initialization, the client is made aware of the server and, similarly, the server is made aware of the client.

[**0065**] To make the server aware of the client, the activation hook on the client can contact the SRM through the LRM to pass it the desired information. The LRM has a mapping of the client's pointer to all the desired information. To make the client aware of the server, the activation hook on the client can record the desired information in the log or pass it to the LRM.

[**0066**] Once both server and client components have been created and properly initialized, each can associate a reference for the other component with the information desirable to re-create the peer.

[0067] During a method call, the client-side logging policy will log information pertaining to the call, including the client identity, server identity, method identity, and arguments to the method. Each method call is stamped with a new logical ID by the client side logging policy. Arguments to the method call sitting on the stack are serialized. The client identity and the method identity, including the logical ID of the method, are passed to the server side policy using the buffer. The server logging policy records this information, including its own identity. The same procedure is executed on the return of the call.

[0068] To log for method invocation and response, the exemplary format of the log record is used as shown in **FIG. 8**. On a per-method basis, the following is desirably logged. (1) identity of the client component comprising the client component class ID (type CLSID); and the client component logical ID; (2) identity of the server component comprising the server component class ID (type CLSID); and the server component logical ID; (3) identify call comprising the interface ID (IID) on which the method was called; method identifier—an index or name for the method; method instance unique identifier—preferably this identifier is unique to the log for the call. This is because it is desirable to uniquely identify the call, which is not always possible using just its name and arguments. If the method was called twice with the same arguments, then the return values might be different based on server state. The argument can be pre-marshal or post-marshal. Pre-marshal permits checks for pointer validity, whereas post- marshal permits pointer invalidity survival.

[0069] Regarding logging for completion/checkpoint, when a component goes stateless, that information is desirably retained in a log record, as this indicates that recovery of the component is no longer needed for the component state, though it still might be used to re-create the component's messages or calls. An exemplary log record for this is shown in **FIG. 9**.

[0070] It is contemplated that the invention can recover any component, whether it is acting in the role of client, server, or both, and that the invention should not be limited to the recovery of components acting in the role of a server responding to requests from other client components.

[0071] The recovery manager responsible for the component, referred to as the component recovery manager (CRM), is identified. The appropriate CRM is then called to recover the failed component. Assuming execution is now taking place on the CRM, it is verified that the component has not already been recovered. The CRM searches an in-memory structure or log. If it is determined that the component has already been recovered, the CRM returns an interface pointer for the already recovered component. Otherwise, the CRM creates a new instance of the failed component by reading the activation log record from the log and replaying the creation call. The state for the component is reinstalled by replaying calls associated with the failed component from the log. A recovery analysis pass may be performed on the log to identify the appropriate log records. Preferably, all references to the component are then updated with the new interface pointer. The CRM will log the recovery of the component on the log, and then return the new component identifier to other recovery managers.

[0072] If recovery is successful, the CRM recovery manager returns a status code indicating success and a new physical identifier for the recovered component (a physical component supporting the virtual persistent component, hence changing the virtual-to-physical mapping of the recovered component) to the error handler. The error handler will update component runtime tables with the new component identifier and return a status code to the component runtime indicating it can resume (retry) method execution; otherwise, the handler will return an error code to the component runtime.

[0073] Embodiments of the present invention are based on an environment mechanism called contexts. A context is a component wrapper mechanism that transparently intercepts object events, such as creation, activation, and method calls. New component services can be introduced to the runtime by implementing "handlers", referred to as policies, for object events and calls, and including them "in the context". However, the invention is applicable to any runtime infrastructure in which interception of interactions between components is possible.

[0074] Automatic recovery is provided by the composition of context policies that work together to log component state and method calls, and in the event of system failure, to mask the failure from the application and initiate recovery actions to reconstruct impacted components.

[0075] The various techniques described herein may be implemented with hardware or software or, where appropriate, with a combination of both. Thus, the methods and apparatus of the present invention, or certain aspects or portions thereof, may take the form of program code (i.e., instructions) embodied in tangible media, such as floppy diskettes, CD-ROMs, hard drives, or any other machine-readable storage medium, wherein, when the program code is loaded into and executed by a machine, such as a computer, the machine becomes an apparatus for practicing the invention. One or more programs are preferably implemented in a high level procedural or object oriented programming language to communicate with a computer system. However, the program(s) can be implemented in assembly or machine language, if desired. In any case, the language may be a compiled or interpreted language, and combined with hardware implementations.

[0076] The methods and apparatus of the present invention may also be embodied in the form of program code that is transmitted over some transmission medium, such as over electrical wiring or cabling, through fiber optics, or via any other form of transmission, wherein, when the program code is received and loaded into and executed by a machine, such as an EPROM, a gate array, a programmable logic device (PLD), a client computer, a video recorder or the like, the machine becomes an apparatus for practicing the invention. When implemented on a general-purpose processor, the program code combines with the processor to provide a unique apparatus that operates to perform the versioning functionality of the present invention.

[0077] It is noted that the foregoing examples have been provided merely for the purpose of explanation and are in no way to be construed as limiting of the present invention. While the invention has been described with reference to various embodiments, it is understood that the words which have been used herein are words of description and illustration, rather than words of limitations. Further, although the invention has been described herein with reference to

particular means, materials and embodiments, the invention is not intended to be limited to the particulars disclosed herein; rather, the invention extends to all functionally equivalent structures, methods and uses, such as are within the scope of the appended claims.

What is claimed is:

1. A method of providing recovery in a computer system, comprising:

intercepting events comprising at least one of a plurality of method calls, component activations, and component deactivations;

logging the events during execution of stateful piece-wise deterministic (PWD)-components having a component state;

detecting a failure of one of the components based on an error detected during execution; and

replaying the logged events to recover the component state.

2. The method of claim 1, further comprising:

intercepting non-deterministic events;

providing a policy mechanism that, upon interception of the events, permits the logging of the intercepted events;

providing an error handler that is invoked when an error is detected, such error handler being capable of masking the error from the component.

3. The method of claim 1, further comprising:

logging information about how the component is mapped to a first set of resources such that the component can be re-mapped to a different set of resources should a system failure result in a termination of the first set of resources, said re-mapping enabling scalability and load balancing.

4. The method of claim 3, wherein the information includes a logical identifier for the component that identifies logged state information for the component.

5. The method of claim 1, further comprising exactly-once execution comprising:

uniquely identifying messages at a sender;

detecting an original message and at least one duplicate message at a receiver;

eliminating the at least one duplicate message; and

providing a reply to the at least one duplicate message that is the same as a reply to the original message.

6. The method of claim 1, further comprising:

notifying an error handler when the error is detected;

contacting a recovery manager to recover the component that has failed;

re-trying an interaction that failed at the time of the failure; and

returning the result of the re-tried interaction to the component.

7. (canceled)

8. The method of claim 1, further comprising:

retrieving information related to the failed call; and

identifying the component based on the retrieved information.

9. The method of claim 1, further comprising:

updating references to the component with an interface pointer corresponding to the component; and

logging the recovery of the component.

10. (canceled)

11. The method of claim 1, further comprising intercepting the method call and response, and invoking a policy on at least one of the call and the response.

12. The method of claim 1, wherein the computer system comprises a client and a server, and detecting the failure of the component comprises the client or server detecting the failure, and further comprising a component runtime associated with the client invoking an error handler associated with the client and providing information related to the failed call to the error handler.

13. The method of claim 12, further comprising identifying a recovery manager responsible for the component and passing the information to the recovery manager.

14. (canceled)

15. A system for providing recovery in a computer system, comprising:

a storage device that logs events comprising at least one of a plurality of method calls, component activations, and component deactivations during execution of stateful piece-wise deterministic (PWD) components having a component state;

a component runtime that receives the events from the storage device;

a error handler that receives the events from the component runtime and identifies the component based on the events; and

a recovery manager that receives the events from the storage device and replays the events to transparently and automatically recover the component state.

16. The system of claim 15, wherein the error handler is notified of an error and contacts the recovery manager to recover the component state.

17. (canceled)

18. The system of claim 15, wherein the component runtime intercepts the at least one method call and invokes a policy on the at least one method call.

19. The system of claim 15, further comprising an activation hook that logs information in the storage device that is sufficient to re-create the component.

20-23. (canceled)

24. A computer readable medium having computer-executable instructions for performing the steps comprising:

intercepting events comprising at least one of a plurality of method calls, component activations, and component deactivations;

logging the events during execution of at least one stateful piece-wise deterministic (PWD) component having a component state in a computer system;

detecting a failure of one of the components based on an error detected during execution; and

replaying the logged events to transparently and automatically recover the component state.

25. The computer readable medium of claim 24, having further computer-executable instructions for:

intercepting non-deterministic events;

providing a policy mechanism that, upon interception of the events, permits the logging of the intercepted events;

providing an error handler that is invoked when an error is detected, such error handler being capable of masking the error from the component.

26. The computer readable medium of claim 24, having further computer-executable instructions for:

logging information about how the component is mapped to a first set of resources such that the component can be re-mapped to a different set of resources should a system failure result in a termination of the first set of resources, said re-mapping enabling scalability and load balancing.

27. The computer readable medium of claim 26, wherein the information includes a logical identifier for the component that identifies logged state information for the component.

28. The computer readable medium of claim 24, having further computer-executable instructions for providing exactly-once execution comprising:

uniquely identifying messages at a sender;

detecting an original message and at least one duplicate message at a receiver;

eliminating the at least one duplicate message; and

providing a reply to the at least one duplicate message that is the same as a reply to the original message.

29. The computer readable medium of claim 24, having further computer-executable instructions for:

notifying an error handler when the error is detected;

contacting a recovery manager to recover the component that has failed;

re-trying an interaction that failed at the time of the failure; and

returning the result of the re-tried interaction to the component.

30. (canceled)

31. The computer readable medium of claim 24, having further computer-executable instructions for:

retrieving information related to the failed call; and

identifying the component based on the retrieved information.

32. The computer readable medium of claim 24, having further computer-executable instructions for:

updating references to the component with an interface pointer corresponding to the component; and

logging the recovery of the component.

33. (canceled)

* * * * *