US 20190332518A1

(19) United States

(12) Patent Application Publication (10) Pub. No.: US 2019/0332518 A1

Lukman et al. (43) Pub. Date: Oct. 31, 2019

(54) MODEL CHECKER FOR FINDING DISTRIBUTED CONCURRENCY BUGS

(71) Applicants: **Futurewei Technologies, Inc.**, Plano, TX (US); **University of Chicago**, Chicago, IL (US)

(72) Inventors: **Jeffrey Lukman**, Chicago, IL (US); **Huan Ke**, Chicago, IL (US); **Haryadi Gunawi**, Chicago, IL (US); **Feng Ye**, Mississauga (CA); **Chen Tian**, Union City, CA (US); **Shen Chi Chen**, San Jose, CA (US)

(21) Appl. No.: **15/962,873**

(22) Filed: **Apr. 25, 2018**

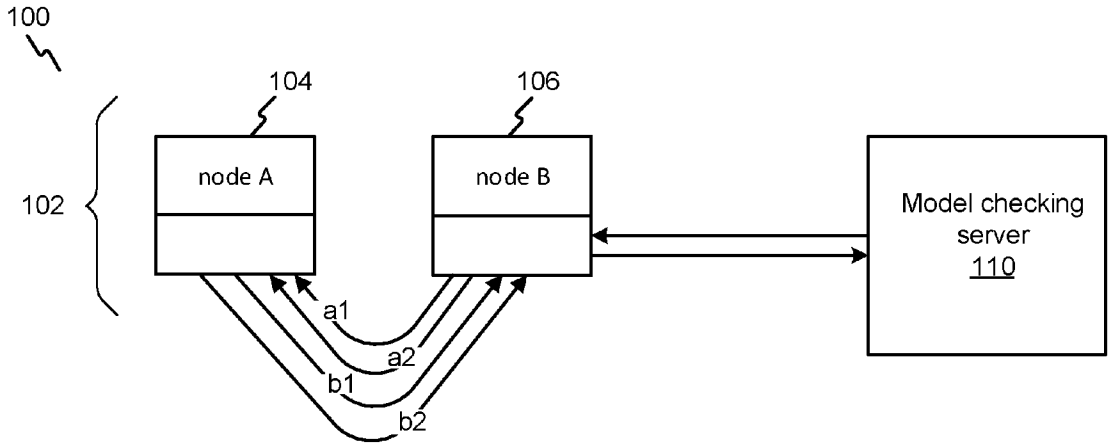
Publication Classification

(51) **Int. Cl.**
G06F 11/36 (2006.01)

(52) **U.S. Cl.**
CPC ***G06F 11/3632*** (2013.01); ***G06F 11/3692*** (2013.01); ***G06F 11/3688*** (2013.01)

(57) **ABSTRACT**

Described herein are systems and methods for distributed concurrency (DC) bug detection. The method includes identifying a plurality of nodes in a distributed computing cluster; identifying a plurality of messages to be transmitted during execution of an application by the distributed computing cluster; determining a set of orderings of the plurality of messages for DC bug detection, the set of orderings determined based upon the plurality of nodes and the plurality of messages; removing a subset of the orderings from the set of orderings based upon one or more of a state symmetry algorithm, a disjoint-update independence algorithm, or a zero-crash-impact reordering algorithm; and performing DC bug detection testing using the set of orderings after the subset of the orderings is removed from the set of orderings.



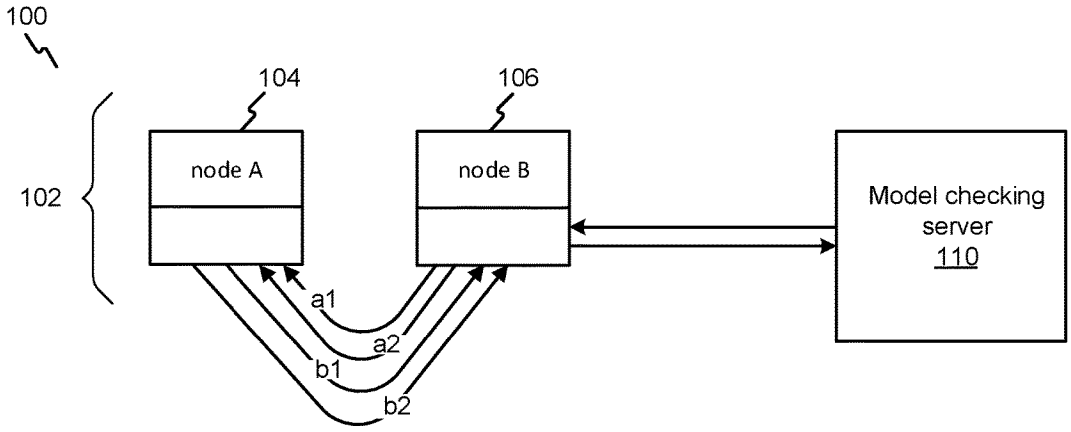

100
104
106
node A
node B
102
Model checking server
110
a1
a2
b1
b2

FIG. 1

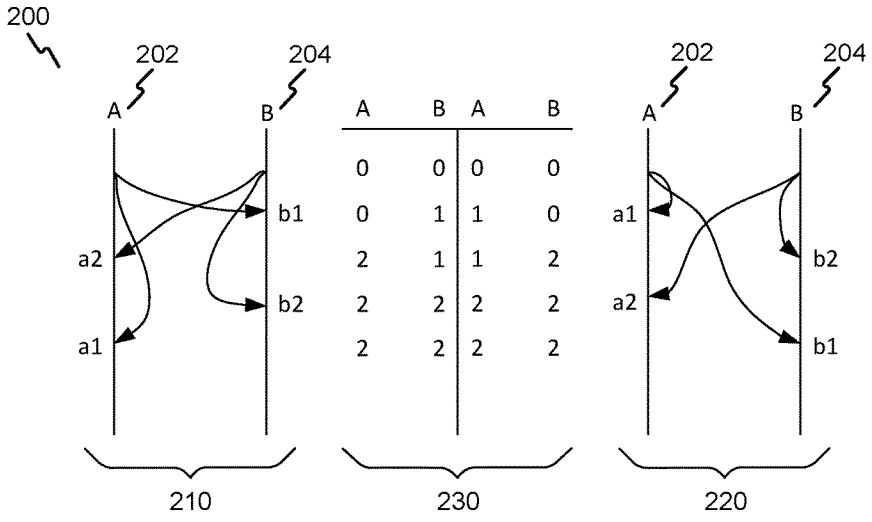

200
202
204
A
B
a1
a2
b1
b2
A B A B
0 0 0 0
0 1 1 0
2 1 1 2
2 2 2 2
2 2 2 2
210
230
220

FIG. 2

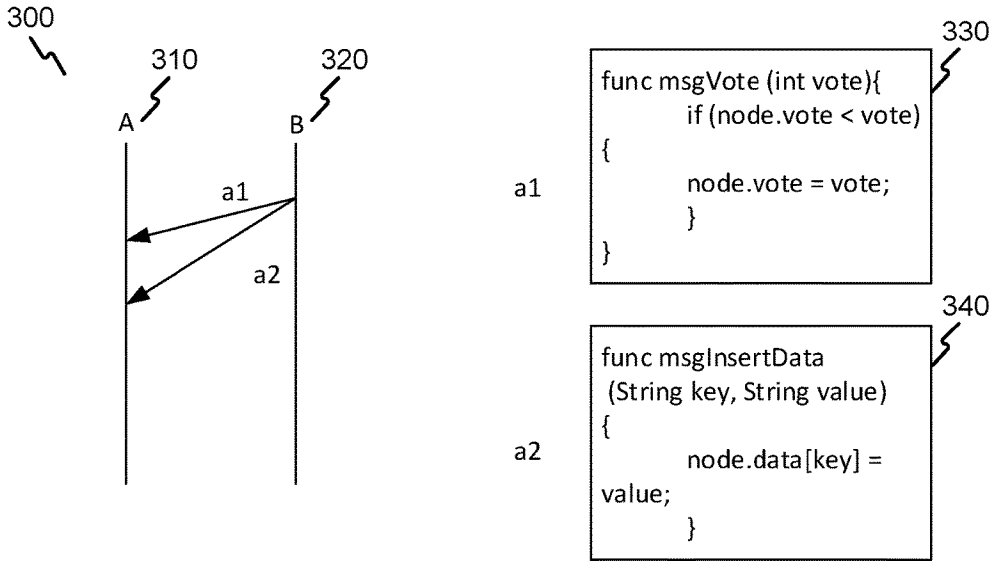

300
310
320
A
B
a1
a2
330
func msgVote (int vote){
if (node.vote < vote)
{
node.vote = vote;
}
}
a1
340
func msgInsertData
(String key, String value)
{
node.data[key] =
value;
}
a2

FIG. 3

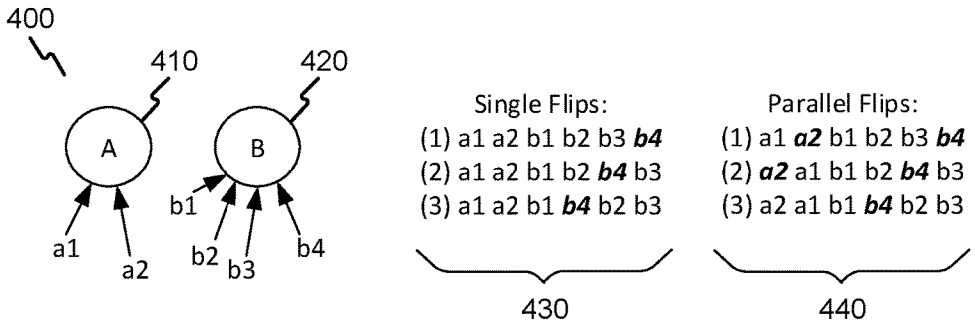

400
410
420
A
B
a1
a2
b1
b2
b3
b4
Single Flips:
(1) a1 a2 b1 b2 b3 b4
(2) a1 a2 b1 b2 b4 b3
(3) a1 a2 b1 b4 b2 b3
Parallel Flips:
(1) a1 a2 b1 b2 b3 b4
(2) a2 a1 b1 b2 b4 b3
(3) a2 a1 b1 b4 b2 b3
430
440

FIG. 4

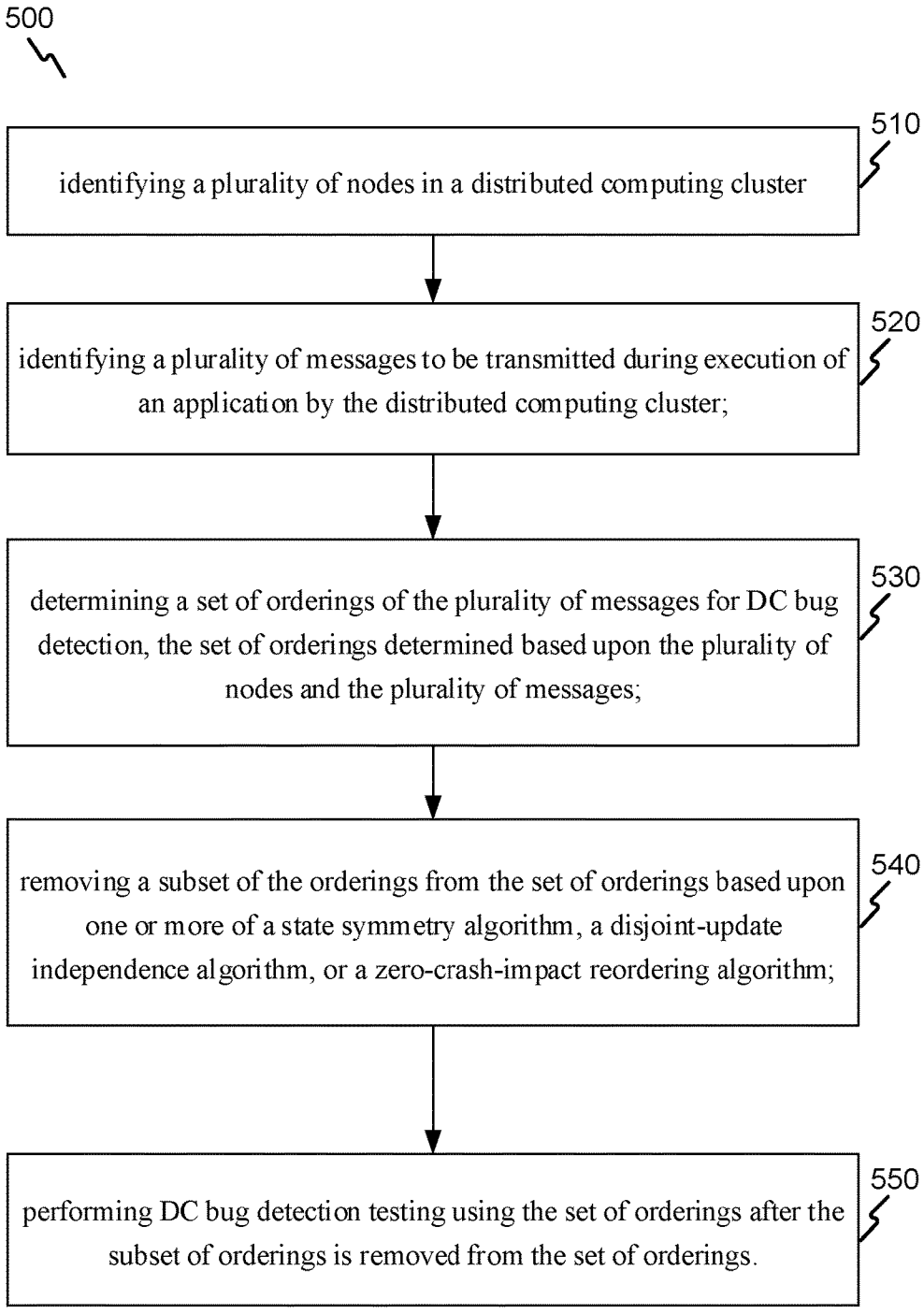

500
identifying a plurality of nodes in a distributed computing cluster
510
identifying a plurality of messages to be transmitted during execution of an application by the distributed computing cluster;
520
determining a set of orderings of the plurality of messages for DC bug detection, the set of orderings determined based upon the plurality of nodes and the plurality of messages;
530
removing a subset of the orderings from the set of orderings based upon one or more of a state symmetry algorithm, a disjoint-update independence algorithm, or a zero-crash-impact reordering algorithm;
540
performing DC bug detection testing using the set of orderings after the subset of orderings is removed from the set of orderings.
550

FIG. 5

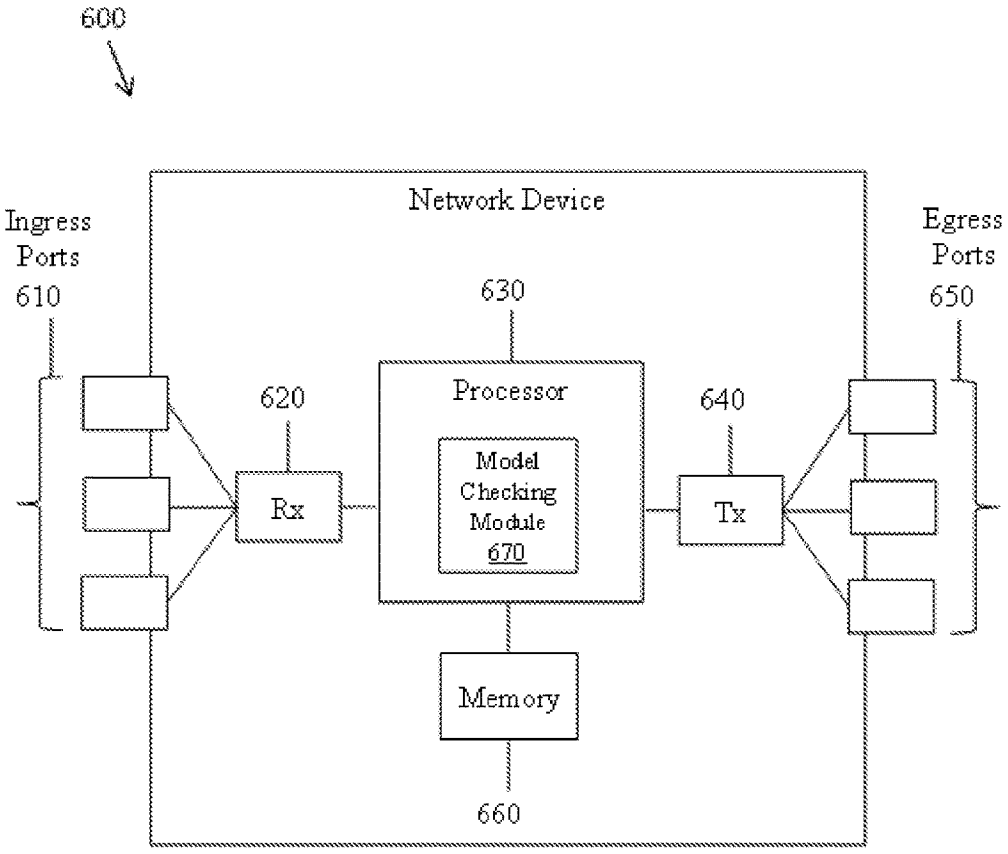

600
Network Device
Ingress Ports
610
Egress Ports
650
630
Processor
620
640
Rx
Model Checking Module
670
Tx
Memory
660

FIG. 6

MODEL CHECKER FOR FINDING DISTRIBUTED CONCURRENCY BUGS

TECHNICAL FIELD

[0001] The disclosure is related to the technical field of distributed computing, in particular detection of distributed concurrency bugs in a distributed computing system.

BACKGROUND

[0002] Cloud computing systems such as distributed computing frameworks, storage systems, lock services, and cluster managers are the backbone engines of many software based applications. Cloud computing systems typically include many nodes physically distributed and connected via a network, e.g., the Internet. The nodes store, manage, and process data. Groups of nodes are often referred to as clusters. The complexities and intricacies of the cloud computing systems make them difficult to manage. One issue is the problem of distributed concurrency (DC) bugs which are caused by concurrent distributed events occurring in a nondeterministic order. DC bugs can cause harmful consequences in cloud computing systems including system crashes, failed jobs, node/cluster unavailability, data loss, and data inconsistency. For example, a cloud computing system is configured to transmit messages A, B, and C to or from one of nodes 1, 2, and 3. The messages are transmitted in response to completion of a task or operation at the node that transmits the message. When node 2 receives message A, node 3 receives message B, and then node 2 receives message C from node 3, the system functions as expected. When the ordering of the messages is changed, e.g., node 3 receives message B and then transmits message C to node 2 prior to node 2 receiving message A from node 1, a failure will happen at node 2. A DC bug has occurred by changing the order of the messages received at node 2.

SUMMARY

[0003] In an embodiment, the disclosure includes a method for distributed concurrency (DC) bug detection. The method includes identifying, by a computing device, a plurality of nodes in a distributed computing cluster; identifying, by the computing device, a plurality of messages to be transmitted during execution of an application by the distributed computing cluster; determining, by the computing device, a set of orderings of the plurality of messages for DC bug detection, the set of orderings determined based upon the plurality of nodes and the plurality of messages; removing, by the computing device, a subset of the orderings from the set of orderings based upon one or more of a state symmetry algorithm, a disjoint-update independence algorithm, or a zero-crash-impact reordering algorithm; and performing, by the computing device, DC bug detection testing using the set of orderings after the subset of the orderings is removed from the set of orderings.

[0004] Optionally, in any of the preceding aspects, removing the subset of the orders from the set of orderings based upon the state symmetry algorithm comprises includes comparing a first state transition of a first node of a first ordering of the set of orderings with a second state transition of a second node of a second ordering of the set of orderings; and adding the second ordering to the subset of the orderings when the first state transition and the second state transition are symmetrical.

[0005] Optionally, in any of the preceding aspects, removing the subset of the orders from the set of orderings based upon the disjoint-update independence algorithm includes comparing a first variable in a first message of a first ordering of the set of orderings with a second variable in a second message of the first ordering of the set of orderings; and adding a second ordering to the subset of the orderings when the first variable and the second variable are different and the second ordering comprises the first message and the second message.

[0006] Optionally, in any of the preceding aspects, the method further includes determining, prior to performing the DC bug detection, one or more parallel flip orderings, each of the parallel flip orderings comprising a first plurality of messages for a first node and a second plurality of messages for a second node, wherein the first plurality of messages are independent of the second plurality of messages, and wherein the first plurality of messages and the second plurality of messages are reordered in each of the parallel flip orderings; and prioritizing the parallel flip orderings when performing the DC bug detection.

[0007] Optionally, in any of the preceding aspects, the zero-crash-impact reordering algorithm includes a crash-after-discard reduction or a consecutive-crash reduction.

[0008] Optionally, in any of the preceding aspects, removing the subset of the orders from the set of orderings based upon crash-after-discard reduction includes determining a first message of a first ordering will be discarded by a node; determining a second message of the first ordering causes a crash of the node; and adding a second ordering comprising the first message and the second message to the subset of the orderings.

[0009] Optionally, in any of the preceding aspects, removing the subset of the orders from the set of orderings based upon consecutive-crash reduction includes determining a first message of a first ordering causes a crash of a node; determining a second message of the first ordering causes another crash of the node; and adding a second ordering comprising the first message and the second message to the subset of the orderings.

[0010] Optionally, in any of the preceding aspects, the set of orderings includes unique orderings for each permutation of the plurality of messages received at each of the plurality of nodes.

[0011] Optionally, in any of the preceding aspects, the method further includes determining the subset of the orderings based upon each of the state symmetry algorithm, the disjoint-update independence algorithm, the zero-crash-impact reordering algorithm, and a parallel flips algorithm.

[0012] In an embodiment, the disclosure includes a device. The device includes a memory storage comprising instructions; and a processor in communication with the memory. The processor executes the instructions to identify a plurality of nodes in a distributed computing cluster; identify a plurality of messages to be transmitted during execution of an application by the distributed computing cluster; determine a set of orderings of the plurality of messages for distributed concurrency (DC) bug detection, the set of orderings determined based upon the plurality of nodes and the plurality of messages; remove a subset of the orderings from the set of orderings based upon one or more of a state symmetry algorithm, a disjoint-update independence algorithm, or a zero-crash-impact reordering algorithm; and

perform DC bug detection testing using the set of orderings after the subset of the orderings is removed from the set of order.

[0013] Optionally, in any of the preceding aspects, the instructions to remove the subset of the orders from the set of orderings based upon the state symmetry algorithm include instructions to compare a first state transition of a first node of a first ordering of the set of orderings with a second state transition of a second node of a second ordering of the set of orderings; and add the second ordering to the subset of the orderings when the first state transition and the second state transition are symmetrical.

[0014] Optionally, in any of the preceding aspects, the instructions to remove the subset of the orders from the set of orderings based upon the disjoint-update independence algorithm include instructions to compare a first variable in a first message of a first ordering of the set of orderings with a second variable in a second message of the first ordering of the set of orderings, and add a second ordering to the subset of the orderings when the first variable and the second variable are different and the second ordering comprises the first message and the second message.

[0015] Optionally, in any of the preceding aspects, the processor further executes the instructions to determine, prior to performing the DC bug detection, one or more parallel flip orderings, each of the parallel flip orderings comprising a first plurality of messages for a first node and a second plurality of messages for a second node, wherein the first plurality of messages are independent of the second plurality of messages, and wherein the first plurality of messages and the second plurality of messages are reordered in each of the parallel flip orderings, and prioritize the parallel flip orderings when performing the Dc bug detection.

[0016] Optionally, in any of the preceding aspects, the zero-crash-impact reordering algorithm includes a crash-after-discard reduction or a consecutive-crash reduction.

[0017] Optionally, in any of the preceding aspects, instructions to remove the subset of the orders from the set of orderings based upon the crash-after-discard reduction include instructions to determine a first message of a first ordering will be discarded by a node, determine a second message of the first ordering causes a crash of the node, and add a second ordering comprising the first message and the second message to the subset of the orderings.

[0018] Optionally, in any of the preceding aspects, instructions to remove the subset of the orders from the set of orderings based upon the consecutive-crash reduction includes instructions to determine a first message of a first ordering causes a crash of a node, determine a second message of the first ordering causes another crash of the node, and add a second ordering comprising the first message and the second message to the subset of the orderings.

[0019] Optionally, in any of the preceding aspects, the set of orderings includes unique orderings for each permutation of the plurality of messages received at each of the plurality of nodes.

[0020] Optionally, in any of the preceding aspects, the processor further executes the instructions to determine the subset of the orderings based upon each of the state symmetry algorithm, the disjoint-update independence algorithm, the zero-crash-impact reordering algorithm, and a parallel flips algorithm.

[0021] In an embodiment, the disclosure includes a non-transitory computer readable medium storing computer instructions, that when executed by a processor, causes the processor to perform identify a plurality of nodes in a distributed computing cluster; identify a plurality of messages to be transmitted during execution of an application by the distributed computing cluster; determine a set of orderings of the plurality of messages for distributed concurrency (DC) bug detection; remove a subset of the orderings from the set of orderings based upon one or more of a state symmetry algorithm, a disjoint-update independence algorithm, or a zero-crash-impact reordering algorithm; and perform DC bug detection testing using the set of orderings after the subset of the orderings is removed from the set of orderings.

[0022] Optionally, in any of the preceding aspects, the the instructions that cause the processor to remove the subset of the orders from the set of orderings based upon the state symmetry algorithm include instructions that cause the processor to compare a first state transition of a first node of a first ordering of the set of orderings with a second state transition of a second node of a second ordering of the set of orderings, and add the second ordering to the subset of the orderings when the first state transition and the second state transition are symmetrical.

[0023] Optionally, in any of the preceding aspects, the the instructions that cause the processor to remove the subset of the orders from the set of orderings based upon the disjoint-update independence algorithm include instructions that cause the processor to compare a first variable in a first message of a first ordering of the set of orderings with a second variable in a second message of the first ordering of the set of orderings, and add a second ordering to the subset of the orderings when the first variable and the second variable are different and the second ordering comprises the first message and the second message.

[0024] Optionally, in any of the preceding aspects, the instructions further cause the processor to determine, prior to the DC bug detection, one or more parallel flip orderings, each of the parallel flip orderings comprising a first plurality of messages for a first node and a second plurality of messages for a second node, wherein the first plurality of messages are independent of the second plurality of messages, and wherein the first plurality of messages and the second plurality of messages are reordered in each of the parallel flip orderings, and prioritize the parallel flip orderings when performing the DC bug detection.

[0025] Optionally, in any of the preceding aspects, the zero-crash-impact reordering algorithm is a crash-after-discard reduction or a consecutive-crash reduction.

[0026] Optionally, in any of the preceding aspects, the instructions that cause the processor to remove the subset of the orders from the set of orderings based upon the crash-after-discard reduction include instructions that cause the processor to determine a first message of a first ordering will be discarded by a node, determine a second message of the first ordering causes a crash of the node, and add a second ordering comprising the first message and the second message to the subset of the orderings.

[0027] Optionally, in any of the preceding aspects, the instructions that cause the processor to remove the subset of the orders from the set of orderings based upon the consecutive-crash reduction include instructions that cause the processor to determine a first message of a first ordering

causes a crash of a node, determine a second message of the first ordering causes another crash of the node, and add a second ordering comprising the first message and the second message to the subset of the orderings.

[0028] Optionally, in any of the preceding aspects, the set of orderings includes unique orderings for each permutation of the plurality of messages received at each of the plurality of nodes.

[0029] Optionally, in any of the preceding aspects, the instructions further cause the processor to determine the subset of the orderings based upon each of the state symmetry algorithm, the disjoint-update independence algorithm, the zero-crash-impact reordering algorithm, and a parallel flips algorithm.

[0030] For the purpose of clarity, any one of the foregoing embodiments may be combined with any one or more of the other foregoing embodiments to create a new embodiment within the scope of the present disclosure.

[0031] These and other features will be more clearly understood from the following detailed description taken in conjunction with the accompanying drawings and claims.

BRIEF DESCRIPTION OF THE DRAWINGS

[0032] For a more complete understanding of this disclosure, reference is now made to the following brief description, taken in connection with the accompanying drawings and detailed description, wherein like reference numerals represent like parts.

[0033] FIG. **1** is a diagram of an embodiment of DC bug detection architecture.

[0034] FIG. **2** is a diagram of an embodiment of permutations used in state symmetry reductions.

[0035] FIG. **3** is a diagram of an embodiment of disjoint-update independence.

[0036] FIG. **4** is a diagram of an embodiment of parallel flips testing.

[0037] FIG. **5** is a diagram of an embodiment of a method for DC bug detection.

[0038] FIG. **6** is a schematic diagram of a network device according to an embodiment of the disclosure.

DETAILED DESCRIPTION

[0039] It should be understood at the outset that, although an illustrative implementation of one or more embodiments are provided below, the disclosed systems and/or methods may be implemented using any number of techniques, whether currently known or in existence. The disclosure should in no way be limited to the illustrative implementations, drawings, and techniques illustrated below, including the exemplary designs and implementations illustrated and described herein, but may be modified within the scope of the appended claims along with their full scope of equivalents.

[0040] Cloud computing involves performing operations across a network of nodes. The operations may be performed responsive to execution of a software application (or "application"). As used herein, an application includes instructions or operations that will be executed in a cloud based system. Cloud based systems include nodes physically distributed and connected via a network, e.g., the Internet. The nodes of a cloud based system can store, manage, and process data. The data storage, management, and processing capabilities of the nodes of the cloud based system can be shared to perform computing tasks. Instructions or operations of an application executed by a cloud based system may distributed across one or more of the nodes. Cloud based systems include distributed computing frameworks, storage systems, lock services, and cluster managers. When an operation is executed, the state of the node that executes the operation may change. A change in state of the node may occur based upon the operation performed or the current state of the node. In some cases, an operation may not cause the state of the node to change. Other nodes may or may not be aware of the current state of the node that executed the operation. The node that executed the operations may send a message comprising a command or data to a second node. Messages include instructions or operations sent from one node of the cloud based system to another node of the cloud based system. For example, messages can include instructions to update a variable, perform a calculation, or display a variable. Sometimes the second node may crash or experience other performance issues if the message from the node that executed the operation is incompatible with the current state of the second node. As used herein distributed concurrency (DC) bugs may refer to an error resulting from the order and timing transmission and receipt of messages, between two or more nodes in a cloud computing system.

[0041] Embodiments of the present disclosure are directed to methods, systems, and apparatuses for detecting DC bugs in a cloud computing system. In an embodiment, a distributed system model checker may implement algorithms for improving the ability to detect DC bugs. In some embodiments, the algorithms may reduce the search space of testing the permutations of message ordering in a cloud based system. A message ordering includes a time ordered sequence of messages arriving at one or more nodes during execution of an application. Permutations of message orderings includes several message orderings with a varied time sequence of arrival of the messages in each permutation. In some embodiments, the algorithms may prioritize certain permutations to decrease the time required for testing. The algorithms may include a state symmetry algorithm, a disjoint-update independence algorithm, a parallel flips algorithm, and/or a zero-crash-impact reordering algorithm, each of which are described in greater detail herein.

[0042] FIG. **1** is a diagram of an embodiment of DC bug detection architecture **100**. The DC bug detection architecture **100** includes a model checking server **110**, node A **104**, and node B **106**. In other embodiments, more than two nodes may be present in the architecture. The number of nodes depends upon the characteristics of the cloud based system where the application under test is executed. Node A **104** and node B **106** are be grouped as a cluster **102**. By way of illustration, cluster **102** executes an application under test wherein the application can send several messages: a**1**, a**2**, b**1**, and b**2**. While four messages are depicted in this illustration, an application under test may transmit and receive many more messages depending upon the functionality of the application. The number of permutations of messages may be equal to the number of messages factorial. In this illustration, the number of permutations of messages is four factorial or twenty-four possible permutations. The model checking server **110** may enable the messages in each of the possible permutations and monitor the results of the various permutations of messages. Enabling a message may include the model checking server **110** sending a message or the model checking server **110** causing a node to send a

message. The model checking server **110** tracks permutations that have been executed and permutations that are to-be executed. A permutation is considered executed after all of the messages in the permutation have been sent, i.e., enabled, according to the message ordering in the permutation. For permutations that have been executed, the model checking server **110** tracks whether or not there was an error in relation to that particular permutation of messages. While the model checking server **110** is depicted as communicating with node B **106**, model checking server **110** can communicate with all or some of the nodes under test in a distributed computing environment. In some embodiments, algorithms are used to determine that certain permutations need not to be tested. Those algorithms will be discussed in detail below.

[0043] In some embodiments, a state symmetry algorithm can be executed to reduce the number of permutations that need to be tested. The state symmetry algorithm can identify pairs of permutations that result in symmetrical state transitions. For pairs of permutations with symmetrical state transitions, only one of the permutations may need to be tested. FIG. **2** is a diagram of an embodiment of a permutations **200** used in state symmetry reductions. By way of illustration, FIG. **2** represents the first phases of a leader election implementation with two concurrent updates from node A **202** and node B **204**. While the state symmetry algorithm can be used with other distributed computing protocols, leader election is used here as an example to illustrate the state symmetry algorithm. Leader election is a process of designating a node or process as the organizer of a task distributed among multiple nodes, in this case, node A **202** and node B **204**. Node A **202** broadcasts 'prepare' messages a**1** and b**1** while node B **204** broadcasts 'prepare' messages a**2** and b**2**. The messages arrive at their destinations at different times based on a number of factors, e.g., network configuration and/or network loading. Message flow **210** represents a first permutation of message arrivals and message flow **220** represents a second permutation of message arrivals. Table **230** depicts the state of each node after a message is received, e.g., state transition of the nodes when messages are received. The left column of table **230** corresponds to message flow **210** and the right column of table **230** corresponds to message flow **220**. Message flow **210** receives messages at their respective destinations in the following order: b**1**, a**2**, b**2**, a**1**. Message flow **220** receives messages at their respective destinations in the following order: a**1**, b**2**, a**2**, b**1**. The messages may include a ballot number in this example. In the context of leader election, a ballot number is an identifier for a round of a leader election. Outside the context of leader election, a ballot number can be an identifier for a particular process to be distributed in a consensus. Messages with a '1', e.g., a**1**, b**1**, can represent a ballot number of 1. Messages with a '2' e.g., a**2**, b**2**, can represent a ballot number of 2. Each row of table **230** represents a particular time and the states of the nodes with respect to receipt messages a**1**, a**2**, b**1**, and b**2** at that time. The first row represents an initial state where both nodes are all zeroes. At the second row, message flow **210** receives message b**1** at node B **204** and the state on the left column of table **230** is set to zero for node A **202** and one for node B **204**. Also at the second row, message flow **220** receives message a**1** at node A **202** and the state on the right column of table **230** is set to one for node A **202** and zero for node B **204**. At the completion of message flow **210** and message flow **220**, the state changes tracked in table **230** of node A **202** with respect to message flow **210** are the same as the state changes tracked in table **230** of node B **204** for message flow **220**. Likewise, the state changes tracked in table **230** of node B **204** with respect to message flow **210** are the same as the state changes of node A **202** with respect to message flow **220**. Thus, the results are considered to have symmetry and one of the permutations can be omitted from testing in a leader election process. By identifying which permutations result in state symmetry, the identified permutations can be eliminated from testing and the test time will be reduced thusly.

[0044] In further embodiments, a disjoint-update independence algorithm can be utilized to reduce the number of permutations that need to be tested. The disjoint-update independence algorithm detects permutations with messages that update different variables. If the messages update different variables, then testing both permutations may be unnecessary. FIG. **3** is a diagram of an embodiment of disjoint-update independence **300**. Node B **320** concurrently transmits messages a**1** and a**2** to node A **310**. Message content **330** of message a**1** can include a read and write of the 'vote' variable, for example. Message content **340** of message a**2** can include a write of the 'key' variable, for example. The variable updated by message a**1** is different than the variable updated by message a**2**. When messages update unrelated variables, a disjoint-update can occur. When different variables are updated by two messages, the order of arrival of the messages at the node may not be relevant to the final state of the node. For example, the value of 'vote' at node A **310** will have the same final value whether message a**1** is received first or message a**2** is received first. Likewise, the value of variable 'key' at node A **310** will have the same final value whether message a**1** is received first or message a**2** is received first. In this case, message ordering a**1**, a**2** and a**2**, a**1** result in a same final state of node A **310**, thus one of the orderings may be discarded.

[0045] Disjoint-update independence **300** can be further described in light of the following. For messages ni and nj sent to a node N, a static analysis can be used to build live variable sets: readSet, updateSet and persistSet. The static analysis includes identifying variables in the messages of two or more permutations. The readSet includes to-be-read variables in the messages, i.e., variables that will be read when a message is transmitted. The updateSet includes to-be-updated variables in the messages, i.e., variables that will be read when a message is transmitted. The persistSet includes to-be-persisted variables, i.e., variables that will be unchanged when a message is transmitted. The live variable sets reflect changes in ni's and nj's read, update, and send sets as node N transitions to a different state after receiving message ni or nj. Given such information, ni and nj are marked disjoint-update independent if ni's readSet, updateSet, and persistSet do not overlap with nj's updateSet, and vice versa. I.e., nj's udateSet does not reflect an update to any of ni's live variable sets, and vice versa. Thus, the ordering of message ni and nj may have the same result as reordering nj and ni, and one of the orderings may be skipped during testing.

[0046] In further embodiments, a parallel flips algorithm can be used to speed up testing relative to existing model checking systems. The parallel flips algorithm includes identifying independent messages in a permutation involving at least two nodes. The independent messages may be

flipped, e.g., reordered, in parallel for the two or more nodes in a single permutation. FIG. **4** is a diagram of an embodiment of parallel flips testing example test **400**. In this example, node A **410** receives messages a**1** and a**2**, and node B **420** receives messages b**1**, b**2**, b**3**, and b**4**. Single flip orderings **430** represent a portion of the permutations tested in single flip testing of node A **410** and node B **420** with respect to messages a**1**, a**2**, b**1**, b**2**, b**3**, and b**4**. Parallel flip orderings **440** represent a portion of the permutations tested in parallel flip testing of node A **410** and node B **420** with respect to messages a**1**, a**2**, b**1**, b**2**, b**3**, and b**4**. As shown, only one message, b**4**, is flipped (e.g., reordered) from permutation (**1**) to permutation (**2**) in single flips orderings **430**. For parallel flips, two messages, b**4** and a**2**, are flipped (e.g., reordered) from permutation (**1**) to permutation (**2**) in parallel flips orderings **440**. Parallel flips algorithm can speed up testing by flipping pairs of messages that are independent of each other. For example, message a**2** arrives at node A **410** and is independent of message b**4** which arrives at node B **420**. Therefore, the messages can be flipped in parallel rather than one at a time, thereby speeding up the testing of the nodes. Parallel flips orderings can be prioritized over single flips orderings in order to more quickly test the messages. For example, a parallel flip ordering tests two messages arrival at two nodes simultaneously. The same testing using single flips may require at least two testing cycles. In some embodiments, orderings with a single flip that is tested using a parallel flip may be skipped during testing.

[0047] A zero-crash-impact reduction algorithm may be executed to reduce the number of permutations that need to be tested. The zero-crash impact reduction algorithm identifies permutations that result in a crash and removes permutations that include the crash from further testing. Zero-crash-impact reduction includes two cases where certain reorderings that cause a node to crash may be discarded from testing. The two cases may include crash-after-discard reduction and consecutive-crash reduction. Crash-after-discard reduction may include cases where 'mx' is a reordering. Message 'm' may be discarded after received by the node, e.g., message 'm' may not change the state of the node where it is received before being discarded. Message 'x' may be a message that causes a crash on the same node. Reordering is unnecessary as 'm' does not create any state change and 'x' always causes a crash. Hence the reordering 'mx' may be removed. Consecutive-crash reduction may include cases where 'xy' is a reordering, where message 'x' and message 'y' are both crashes. In this case reordering is unnecessary as two consecutive crashes are equivalent to one in terms of system state. Hence reordering 'xy' may be removed from testing.

[0048] FIG. **5** is a diagram of an embodiment of a method **500** for DC bug detection. The method **500** begins at block **510** where a model checking server identifies a plurality of nodes in a distributed computing cluster. The plurality of nodes can be identified using one or more of a number of network discovery techniques. For example, a listing of the nodes can be programmed into the model checking server and/or the model checking server can interact with a networking device to learn the topology of the distributed computing cluster.

[0049] At block **520**, the model checking server identifies a plurality of messages that result from execution of an application by the distributed computing cluster. For example, an application comprises a number of operations that can be performed at one or more of the nodes in the distributed computing cluster. The operations can provide data to other nodes in order to perform a subsequent operation of the application. The data can be provided in messages that are transmitted between nodes.

[0050] At block **530**, the model checking server determines a set of orderings of the plurality of messages for use in DC bug detection. An ordering may be an arrival sequence of the messages of the application at one or more nodes, i.e., a permutation of messages. Each ordering can be a unique sequence of message arrival at one or more of the nodes of the distributed computing cluster. The set of orderings can include all possible sequences of message arrival for each of the plurality of nodes in the distributed computing cluster. By testing all sequences, DC bugs can be detected for sequences that cause performance issues in the distributed computing cluster, e.g., degraded performance and/or node crashes.

[0051] At block **540**, the model checking server removes a subset of the orderings from the set of orderings based upon one or more of a state symmetry algorithm, a disjoint-update independence algorithm, or a zero-crash-impact reordering algorithm. The model checking server executes one or more of the algorithms in order to reduce the number of orderings that need to be tested for DC bug detection. Removing some of the orderings from the set of orderings reduces the testing time required for DC bug detection. As described above, the algorithms can determine orderings that are redundant and don't need to be tested. Optionally at block **540**, the model checking server may execute a parallel flips algorithm to prioritize certain orderings during testing. By prioritizing parallel flip orderings, testing time may be reduced.

[0052] At block **550**, the model checking server performs DC bug detection testing using the set of orderings after the subset of the orderings is removed from the set of orderings. When the set of orderings has been optimized by removing the orderings identified by the algorithms, the testing can be performed with increased efficiency.

[0053] FIG. **6** is a schematic diagram of a network device **600** (e.g., a model checking server) according to an embodiment of the disclosure. The network device **600** is suitable for implementing the disclosed embodiments as described herein. In an embodiment, the network device **600** is a model checking server. The network device **600** comprises ingress ports **610** and receiver units (Rx) **620** for receiving data; a processor, logic unit, or central processing unit (CPU) **630** to process the data; transmitter units (Tx) **640** and egress ports **650** for transmitting the data; and a memory **660** for storing the data. The network device **600** may also comprise optical-to-electrical (OE) components and electrical-to-optical (EO) components coupled to the ingress ports **610**, the receiver units **620**, the transmitter units **640**, and the egress ports **650** for egress or ingress of optical or electrical signals.

[0054] The processor **630** can be implemented by hardware and/or software. The processor **630** can be implemented as one or more CPU chips, cores (e.g., as a multi-core processor), field-programmable gate arrays (FPGAs), application specific integrated circuits (ASICs), and digital signal processors (DSPs). The processor **630** is in communication with the ingress ports **610**, receiver units **620**, transmitter units **640**, egress ports **650**, and memory **660**. The processor **630** comprises a model checking module **670**.

The model checking module **670** implements the disclosed embodiments described above. For instance, the model checking module **670** implements, processes, prepares, or provides the various algorithms described herein. The inclusion of the model checking module **670** therefore provides a substantial improvement to the functionality of the network device **600** and effects a transformation of the network device **600** to a different state. Alternatively, the model checking module **670** is implemented as instructions stored in the memory **660** and executed by the processor **630**.

[0055] The memory **660** comprises one or more disks, tape drives, and solid-state drives and can be used as an over-flow data storage device, to store programs when such programs are selected for execution, and to store instructions and data that are read during program execution. The memory **660** can be volatile and/or non-volatile and can be read-only memory (ROM), random access memory (RAM), ternary content-addressable memory (TCAM), and/or static random-access memory (SRAM).

[0056] A method for distributed concurrency (DC) bug detection including means for identifying a plurality of nodes in a distributed computing cluster; identifying a plurality of messages to be transmitted during execution of an application by the distributed computing cluster; determining a set of orderings of the plurality of messages for DC bug detection, the set of orderings determined based upon the plurality of nodes and the plurality of messages; removing a subset of the orderings from the set of orderings based upon one or more of a state symmetry algorithm, a disjoint-update independence algorithm, or a zero-crash-impact reordering algorithm; and performing DC bug detection testing using the set of orderings after the subset of the orderings is removed from the set of orderings.

[0057] A memory storage means comprising instructions; and a processor means in communication with the memory means. The processor means executes the instructions to identify a plurality of nodes in a distributed computing cluster; identify a plurality of messages to be transmitted during execution of an application by the distributed computing cluster; determine a set of orderings of the plurality of messages for distributed concurrency (DC) bug detection, the set of orderings determined based upon the plurality of nodes and the plurality of messages; remove a subset of the orderings from the set of orderings based upon one or more of a state symmetry algorithm, a disjoint-update independence algorithm, or a zero-crash-impact reordering algorithm; and perform DC bug detection testing using the set of orderings after the subset of the orderings is removed from the set of order.

[0058] A non-transitory computer readable medium means storing computer instructions, that when executed by a processor means, causes the processor means to perform identify a plurality of nodes in a distributed computing cluster; identify a plurality of messages to be transmitted during execution of an application by the distributed computing cluster; determine a set of orderings of the plurality of messages for distributed concurrency (DC) bug detection; remove a subset of the orderings from the set of orderings based upon one or more of a state symmetry algorithm, a disjoint-update independence algorithm, or a zero-crash-impact reordering algorithm; and perform DC bug detection testing using the set of orderings after the subset of the orderings is removed from the set of orderings.

[0059] While several embodiments have been provided in the present disclosure, it should be understood that the disclosed systems and methods might be embodied in many other specific forms without departing from the spirit or scope of the present disclosure. The present examples are to be considered as illustrative and not restrictive, and the intention is not to be limited to the details given herein. For example, the various elements or components can be combined or integrated in another system or certain features can be omitted, or not implemented.

[0060] In addition, techniques, systems, subsystems, and methods described and illustrated in the various embodiments as discrete or separate can be combined or integrated with other systems, modules, techniques, or methods without departing from the scope of the present disclosure. Other items shown or discussed as coupled can be directly coupled or can be indirectly coupled or communicating through some interface, device, or intermediate component whether electrically, mechanically, or otherwise. Other examples of changes, substitutions, and alterations are ascertainable by one skilled in the art and could be made without departing from the spirit and scope disclosed herein.

1. A method for distributed concurrency (DC) bug detection, the method comprising:

identifying, by a computing device, a plurality of nodes in a distributed computing cluster;

identifying, by the computing device, a plurality of messages to be transmitted during execution of an application by the distributed computing cluster;

determining, by the computing device, a set of orderings of the plurality of messages for DC bug detection, the set of orderings determined based upon the plurality of nodes and the plurality of messages;

removing, by the computing device, a subset of the orderings, where each ordering comprises a unique sequence of message arrival at one or more of the nodes, from the set of orderings based upon one or more of a state symmetry algorithm, a disjoint-update independence algorithm, or a zero-crash-impact reordering algorithm; and

performing, by the computing device, DC bug detection testing using the set of orderings after the subset of the orderings is removed from the set of orderings.

2. The method of claim **1**, wherein removing the subset of the orders from the set of orderings based upon the state symmetry algorithm comprises:

comparing a first state transition of a first node of a first ordering of the set of orderings with a second state transition of a second node of a second ordering of the set of orderings; and

adding the second ordering to the subset of the orderings when the first state transition and the second state transition are symmetrical.

3. The method of claim **1**, wherein removing the subset of the orders from the set of orderings based upon the disjoint-update independence algorithm comprises:

comparing a first variable in a first message of a first ordering of the set of orderings with a second variable in a second message of the first ordering of the set of orderings; and

adding a second ordering to the subset of the orderings when the first variable and the second variable are different and the second ordering comprises the first message and the second message.

4. The method of claim 1, further comprising:
determining, prior to performing the DC bug detection, one or more parallel flip orderings, each of the parallel flip orderings comprising a first plurality of messages for a first node and a second plurality of messages for a second node, wherein the first plurality of messages are independent of the second plurality of messages, and wherein the first plurality of messages and the second plurality of messages are reordered in each of the parallel flip orderings; and
prioritizing the parallel flip orderings when performing the DC bug detection.

5. The method of claim 1, wherein the zero-crash-impact reordering algorithm is a crash-after-discard reduction or a consecutive-crash reduction.

6. The method of claim 5, wherein removing the subset of the orders from the set of orderings based upon crash-after-discard reduction comprises:
determining a first message of a first ordering will be discarded by a node;
determining a second message of the first ordering causes a crash of the node; and
adding a second ordering comprising the first message and the second message to the subset of the orderings.

7. The method of claim 5, wherein removing the subset of the orders from the set of orderings based upon consecutive-crash reduction comprises:
determining a first message of a first ordering causes a crash of a node;
determining a second message of the first ordering causes another crash of the node; and
adding a second ordering comprising the first message and the second message to the subset of the orderings.

8. The method of claim 1, wherein the set of orderings comprises unique orderings for each permutation of the plurality of messages received at each of the plurality of nodes.

9. The method of claim 1, further comprising determining the subset of the orderings based upon each of the state symmetry algorithm, the disjoint-update independence algorithm, the zero-crash-impact reordering algorithm, and a parallel flips algorithm.

10. A device comprising:
a memory storage comprising instructions; and
a processor in communication with the memory, wherein the processor executes the instructions to:
identify a plurality of nodes in a distributed computing cluster;
identify a plurality of messages to be transmitted during execution of an application by the distributed computing cluster;
determine a set of orderings of the plurality of messages for distributed concurrency (DC) bug detection, the set of orderings determined based upon the plurality of nodes and the plurality of messages;
remove a subset of the orderings, where each ordering comprises a unique sequence of message arrival at one or more of the nodes, from the set of orderings based upon one or more of a state symmetry algorithm, a disjoint-update independence algorithm, or a zero-crash-impact reordering algorithm; and
perform DC bug detection testing using the set of orderings after the subset of the orderings is removed from the set of orderings.

11. The device of claim 10, wherein the instructions to remove the subset of the orders from the set of orderings based upon the state symmetry algorithm comprise instructions to:
compare a first state transition of a first node of a first ordering of the set of orderings with a second state transition of a second node of a second ordering of the set of orderings; and
add the second ordering to the subset of the orderings when the first state transition and the second state transition are symmetrical.

12. The device of claim 10, wherein the instructions to remove the subset of the orders from the set of orderings based upon the disjoint-update independence algorithm comprise instructions to:
compare a first variable in a first message of a first ordering of the set of orderings with a second variable in a second message of the first ordering of the set of orderings; and
add a second ordering to the subset of the orderings when the first variable and the second variable are different and the second ordering comprises the first message and the second message.

13. The device of claim 10, wherein the processor further executes the instructions to:
determine, prior to performing the DC bug detection, one or more parallel flip orderings, each of the parallel flip orderings comprising a first plurality of messages for a first node and a second plurality of messages for a second node, wherein the first plurality of messages are independent of the second plurality of messages, and wherein the first plurality of messages and the second plurality of messages are reordered in each of the parallel flip orderings; and
prioritize the parallel flip orderings when performing the Dc bug detection.

14. The device of claim 10, wherein the zero-crash-impact reordering algorithm is a crash-after-discard reduction or a consecutive-crash reduction.

15. The device of claim 14, wherein instructions to remove the subset of the orders from the set of orderings based upon the crash-after-discard reduction comprise instructions to:
determine a first message of a first ordering will be discarded by a node;
determine a second message of the first ordering causes a crash of the node; and
add a second ordering comprising the first message and the second message to the subset of the orderings.

16. The device of claim 14, wherein instructions to remove the subset of the orders from the set of orderings based upon the consecutive-crash reduction comprise instructions to:
determine a first message of a first ordering causes a crash of a node;
determine a second message of the first ordering causes another crash of the node; and
add a second ordering comprising the first message and the second message to the subset of the orderings.

17. The device of claim 10, wherein the set of orderings comprises unique orderings for each permutation of the plurality of messages received at each of the plurality of nodes.

18. The device of claim **10**, wherein the processor is further configured to determine the subset of the orderings based upon each of the state symmetry algorithm, the disjoint-update independence algorithm, the zero-crash-impact reordering algorithm, and a parallel flips algorithm.

19. A non-transitory computer readable medium storing computer instructions, that when executed by a processor, causes the processor to perform:

identify a plurality of nodes in a distributed computing cluster;

identify a plurality of messages to be transmitted during execution of an application by the distributed computing cluster;

determine a set of orderings of the plurality of messages for distributed concurrency (DC) bug detection;

remove a subset of the orderings, where each ordering comprises a unique sequence of message arrival at one or more of the nodes, from the set of orderings based upon one or more of a state symmetry algorithm, a disjoint-update independence algorithm, or a zero-crash-impact reordering algorithm; and

perform DC bug detection testing using the set of orderings after the subset of the orderings is removed from the set of orderings.

20. The non-transitory computer readable medium of claim **19**, wherein the instructions that cause the processor to remove the subset of the orders from the set of orderings based upon the state symmetry algorithm comprise instructions that cause the processor to perform:

compare a first state transition of a first node of a first ordering of the set of orderings with a second state transition of a second node of a second ordering of the set of orderings; and

add the second ordering to the subset of the orderings when the first state transition and the second state transition are symmetrical.

21. The non-transitory computer readable medium of claim **19**, wherein the instructions that cause the processor to remove the subset of the orders from the set of orderings based upon the disjoint-update independence algorithm comprise instructions that cause the processor to perform:

compare a first variable in a first message of a first ordering of the set of orderings with a second variable in a second message of the first ordering of the set of orderings; and

add a second ordering to the subset of the orderings when the first variable and the second variable are different and the second ordering comprises the first message and the second message.

22. The non-transitory computer readable medium of claim **19**, wherein the instructions further cause the processor to perform:

determine, prior to the DC bug detection, one or more parallel flip orderings, each of the parallel flip orderings comprising a first plurality of messages for a first node and a second plurality of messages for a second node, wherein the first plurality of messages are independent of the second plurality of messages, and wherein the first plurality of messages and the second plurality of messages are reordered in each of the parallel flip orderings; and

prioritize the parallel flip orderings when performing the DC bug detection.

23. The non-transitory computer readable medium of claim **19**, wherein the zero-crash-impact reordering algorithm is a crash-after-discard reduction or a consecutive-crash reduction.

24. The non-transitory computer readable medium of claim **23**, wherein instructions that cause the processor to remove the subset of the orders from the set of orderings based upon the crash-after-discard reduction comprise instructions that cause the processor to perform:

determine a first message of a first ordering will be discarded by a node;

determine a second message of the first ordering causes a crash of the node; and

add a second ordering comprising the first message and the second message to the subset of the orderings.

25. The non-transitory computer readable medium of claim **23**, wherein instructions that cause the processor to remove the subset of the orders from the set of orderings based upon the consecutive-crash reduction comprise instructions that cause the processor to perform:

determine a first message of a first ordering causes a crash of a node;

determine a second message of the first ordering causes another crash of the node; and

add a second ordering comprising the first message and the second message to the subset of the orderings.

26. The non-transitory computer readable medium of claim **19**, wherein the set of orderings comprises unique orderings for each permutation of the plurality of messages received at each of the plurality of nodes.

27. The non-transitory computer readable medium of claim **19**, wherein the instructions further cause the processor to determine the subset of the orderings based upon each of the state symmetry algorithm, the disjoint-update independence algorithm, the zero-crash-impact reordering algorithm, and a parallel flips algorithm.

* * * * *