

1 **MUTABLE CHRONOLOGIES FOR ACCOMMODATION OF**
2 **RANDOMLY OCCURRING EVENT DELAYS**

3 **CROSS-REFERENCE TO RELATED APPLICATIONS**

4 This application claims the benefit of the July 25, 2014 priority date of U.S.
5 Application No. 62/028,999.

6 **FIELD OF DISCLOSURE**

7 This disclosure relates to event processing, and in particular, to updating
8 variables in response to events.

9 **BACKGROUND**

10 A typical computer system receives events and updates a state in response to
11 those events.

12 A state may consist of a set of state variables that represent account balances
13 for various entities. In that case, an event may be a withdrawal or transfer made by
14 one entity. In response to an event, it may be necessary to change the state by
15 updating one or more of the state variables.

16 In many cases, the state is deterministic. This means that if all events prior to a
17 certain time are known, and all those events are processed in the correct order, the
18 state will be known.

19 Unfortunately, there is no guarantee that events will arrive in the correct order.
20 As a result, the recorded state, which is what the computer believes the state to be,
21 will not always match the actual state, which is what the computer would have
22 recorded if it had had received the relevant events in the correct order.

23 In complex event processing (CEP) systems, for example, streams of events
24 are processed, while, concurrently, actions are taken based on the results. A set of
25 working data for a CEP system may include different state variables that are being
26 operated on based on different respective streams of received event data (e.g., event
27 data such as database requests, weather data, or traffic data). The order of the original
28 sequence of individual events may have an effect on the processing to be performed.
29 In some cases, the state reflected by the state variables may be incorrect if the
30 processing of events occurs out of order.

1 The process of correcting the state is not so simple, particularly in a multi-core
2 or multi-node environment. In many cases, updating a state will involve updating
3 multiple state variables in working storage. This raises the risk of deadlock between
4 two different update computations that depend on each other to make forward
5 progress. To alleviate this deadlock risk, some systems use a form of concurrency
6 control (e.g., pessimistic concurrency control using locks and two-phase commit, or
7 optimistic concurrency control without locks but with a non-local verification
8 procedure). In addition, the overall system should be resilient enough to at least
9 survive failure of a node or core, and to recover the state after such a failure.

10 Conventional approaches to event processing involve a fault-tolerant
11 distributed database that incorporates complex and time-consuming distributed
12 algorithms. Building such a system and making it perform well (e.g., with low
13 latency) at high peak levels of service is generally difficult. For example, making it
14 perform at the level of millions of events per second with sub-millisecond
15 responsiveness is a daunting prospect to say the very least.

16 **SUMMARY**

17 In one aspect, in general, a method for causing a computing system to process
18 events from a sequence of events that defines a correct order for said events
19 independent from an order in which those events are received over an input device or
20 port includes: defining a first variable, defining, for the first variable, a first
21 chronology of operations on the first variable that are associated with received events,
22 receiving a first event that pertains to the first variable, executing a first operation on
23 said first variable, which results in a first update of the first chronology, after having
24 received the first event, receiving a delayed event that pertains to the first variable,
25 executing a second operation on said first variable, which results in a second update
26 of the first chronology, and determining whether the first update is valid or invalid,
27 wherein the delayed event precedes the first event in the sequence, wherein the first
28 update is based on the first event, and wherein the second update is based on the
29 delayed event.

30 In some practices, determining whether the first update is valid or invalid
31 includes determining that the first update is valid. Among these practices are those
32 practices in which determining that the first update is valid includes determining that

1 the first update is the insertion of a write operation before the first update, those
2 practices in which determining that the first update is valid includes determining that
3 the delayed event resulted in insertion of a read operation before the first update, and
4 those in which determining that the first update is valid includes determining that the
5 delayed event required insertion of an update operation before the first update.

6 In some practices of the method, determining that the first update is valid
7 includes determining that the first update is not affected by the second update.

8 In some practices, determining whether the first update is valid or invalid
9 includes determining that the first update is affected by the second update, and
10 rescinding the first update in response to determining that said first update is invalid.

11 In other practices, determining whether the first update is valid or invalid
12 includes determining that the first update is affected by the second update, causing
13 said first update to be invalid. Among these practices are those in which determining
14 that the first update is affected by the second update includes determining that the
15 delayed event required insertion of a chronology entry indicating an operation that
16 includes writing said first variable before a chronology entry corresponding to the first
17 update, and those in which determining that the first update is affected by the second
18 update includes determining that the delayed event required updating an existing
19 chronology entry indicating an operation on said first variable before a chronology
20 entry corresponding to the first update.

21 In some cases in which the correct order for the sequence of events is partially
22 ordered, practices of the method include causing the events to be completely ordered.

23 In alternative practices, receiving the first event includes receiving the event
24 from a set of completely ordered events.

25 Among the practices of the method are those that also include detecting a loss
26 of data in the first chronology (e.g., a portion of the first chronology stored on a
27 device or node that experienced a failure), retrieving a first checkpoint value
28 indicative of a valid value of the first variable in the first chronology, based on the
29 checkpoint value, retrieving a subset of events from a list of events, the list of events
30 including the first event, reprocessing the subset of events, thereby reconstructing the

1 data lost in the loss of data, and revising the first chronology to include the data lost in
2 the loss of data.

3 In yet other practices, the first chronology includes a plurality of entries
4 corresponding to operations resulting from a plurality of events that are after an entry
5 corresponding to a delayed event operation corresponding to the delayed event. Some
6 of these practices include those in which executing a second operation that causes a
7 second update of the first chronology includes identifying a barrier operation that
8 blocks propagation of invalidity in the plurality of entries, and revising only those
9 operations from the plurality of operations whose entries are between entries
10 corresponding to the delayed event operation and the barrier operation. Among these
11 practices are those in which identifying a barrier operation includes identifying a write
12 operation as a barrier operation.

13 In alternative practices, determining whether the first update is valid includes
14 identifying an access to the first chronology that must have occurred to correctly
15 execute the first update, inspecting the first chronology, and based on the inspection,
16 determining whether the access has occurred. Among these practices are those that
17 further include maintaining first information, which identifies an access to the first
18 chronology, and second information, which identifies a range of accesses required to
19 process the first event, and determining that the first information is within the range of
20 accesses.

21 In other practices, the computing system includes a plurality of nodes in
22 communication over a network, and the method further includes causing the
23 computing system to execute the steps of storing the first variable and the first
24 chronology at a first node; storing a second variable and a second chronology of
25 operations on the second variable at a second node; and processing at least some
26 events of the sequence of events by performing operations on both the first and
27 second variables, and updating both of the first and second chronologies in response
28 to processing the events.

29 Other practices include compatible combinations of any of the foregoing
30 practices.

1 In another aspect, the invention features software, stored in non-transitory
2 form on a computer-readable medium, for causing a computing system to process
3 events from a sequence of events that defines a correct order for said events
4 independent from an order in which those events are received over an input device or
5 port, the software including instructions for causing a computing system to carry out
6 any of the foregoing methods.

7 In yet another aspect, the invention features a computing system for
8 processing events from a sequence of events that defines a correct order for said
9 events independent from an order in which those events are received over an input
10 device or port, the computing system including: an input device or port configured to
11 receive said events, and at least one processor configured to participate in processing
12 of the sequence of events, the processing including any of the foregoing methods.

13 In yet another aspect, the invention features a computing system for
14 processing events from a sequence of events that defines a correct order for said
15 events independent from an order in which those events are received over an input
16 device or port, the computing system including: means for receiving said events, and
17 means for carrying out any of the foregoing recited steps.

18 Aspects can have one or more of the following advantages.

19 The techniques described herein are based on adding an explicit temporal
20 dimension to all computations but nevertheless de-coupling this temporal dimension
21 from the order in which the computations are performed. The result is a system that
22 incrementally builds up a chronology of entries (called “frames”) indicating
23 operations performed in responds to events and consequent state changes. As the
24 chronology is updated in response to received events, the chronology eventually
25 converges to a deterministic final state. Eventually, once all events have been received
26 and all computations resulting from those events have been completed, the
27 chronology will have converged to a unique chronology. For example, the system
28 may determine that a sequence of events has finished based on known limits on a
29 delay interval by which an event could be delayed. If there is a point at which no
30 additional events will arrive, the system converges to a deterministic final state. But it
31 may do so by following a path that is not necessarily deterministic.

1 In some embodiments, the need for expensive (in terms of computing time
2 and/or resources) concurrency control procedures can be avoided. For example,
3 procedures like two-phase commit are not needed. Additionally, there is no need for a
4 non-local verification procedure that requires communication between remote nodes.
5 Instead, verification can be achieved using only local processing. Additionally, a
6 recovery procedure is described in which correct local state can be restored after
7 failure of a node without necessarily having to reprocess all stored events. Updates of
8 local state after a delayed event can also be performed without necessarily having to
9 reprocess all stored events whose order is after the delayed event.

10 **BRIEF DESCRIPTION OF THE FIGURES**

11 FIG. 1 shows an event-processing apparatus;
12 FIG. 2 shows additional fields that make up an event;
13 FIG. 3 shows an abbreviated and expanded representation of a chronology;
14 FIGS. 4A-4B show examples of significant and insignificant operations on a
15 chronology;
16 FIG. 5 illustrates steps in execution of the method when events arrive in the correct
17 order;
18 FIG. 6 illustrates steps in execution of the method when events arrive out of order;
19 FIG. 7 illustrates a more complicated example similar to that in FIG. 5;
20 FIG. 8 illustrates a more complicated example similar to that in FIG. 6;
21 FIG. 9 shows why the implementation does not require latches;
22 FIG. 10 shows reconstruction of a chronology;
23 FIG. 11 shows the use of generation counters;
24 FIG. 12 shows the possible states of chronologies for the activity described in FIG.
25 11;

1 FIG. 13 shows the possible states for event frames for the activity described in FIG.
2 11; and

3 FIG. 14 shows examples of regression based on the states in FIGS. 12 and 13.

4 **DETAILED DESCRIPTION**

5 Referring to FIG. 1, an event ingestor **12** has a connection to a network **14**.
6 Various event sources **16, 18, 20** are also connected to the network **14**. Although only
7 one event ingestor **12** is shown, more than one such machine can be provided. In
8 general, each event ingestor **12** can access not only its own locally stored data but data
9 stored by another event ingestor by exchanging messages over the network **14** as
10 needed. In some embodiments, a multi-node computing system includes various
11 nodes that are each configured to include one or more event ingestors and/or one or
12 more event sources. In some embodiments, each event ingestor is implemented on its
13 own node in the network **14**, physically or geographically separated from each other
14 node but in communication over some path through the network **14**.

15 **Surrogate timestamps**

16 Each event source **16, 18, 20** produces events. The network **14** transports these
17 events from the event source **16, 18, 20** to an event ingestor **12**, for example, by
18 transmitting (or broadcasting) event data over the network **14**. When event data is
19 received at the event ingestor **12**, it may be recognized as representing an event that is
20 to be handled by an event-handler. For example, an event-handler may perform a
21 calculation in response to the event, which may involve performing operations on one
22 or more variables representing a working state, as described in more detail below.

23 The events are ordered. In some examples the ordering is temporal, where
24 each event occurs at a particular point in time. That time of occurrence can be
25 represented in a variety of ways. For example, the time of occurrence for an event
26 may correspond to a value of a timestamp included with the event data. If no two
27 events occur at the exact same time, the resulting event sequence is completely
28 ordered.

29 As a practical matter, it is possible for two events to be marked as having
30 occurred at the same time even if there is some small difference in their actual times
31 of occurrence. A common way this can arise is if two events occur closer together

1 than the granularity with which time is measured. In this case, the sequence will be
2 partially ordered.

3 For an event ingestor **12** to implement the method described herein, the event
4 sequence must be completely ordered, not just partially ordered. One method of
5 converting a partially ordered sequence into a completely ordered sequence is to
6 provide a surrogate timestamp to function as a tiebreaker.

7 One method of assigning a surrogate timestamp is to maintain a set of
8 counters, each one of which corresponds to an event source **16, 18, 20**. Each counter
9 is then initialized to a unique seed. In some examples, the seed is a positive integer
10 that is less than the number of event sources. For example, for three event sources, the
11 seed may be “1” or “2.”

12 When an event source **16** generates a new event, it increments its counter by
13 the number of event sources. The event source **16** then tags the event with a surrogate
14 timestamp that includes the original timestamp for the event with the incremented
15 value of the counter appended to that timestamp. Only then does the event source **16**
16 send the new event on to the event ingestor **12**. This process ensures that the event
17 ingestor **12** receives only a set of completely ordered events. To simplify exposition,
18 throughout the remainder of this disclosure, it will be assumed that the surrogate
19 timestamp is simply an integer value of such a counter, without explicitly showing the
20 original timestamp value (which still guarantees a completely ordered sequence of
21 events).

22 In other examples, instead of a temporal ordering, the events may be
23 associated with some other type of quantitative metric that that determines a
24 completely ordered sequence of events, or a partially ordered sequence of events that
25 can be converted into a completely ordered sequence using a tiebreaker.

26 **Saving an event**

27 After an event has been received at the event ingestor **12** and given a surrogate
28 timestamp, it is retained and made durable. This can be carried out in a variety of
29 ways. For example, the event can be written to a durable storage medium, such as a
30 hard disk drive (or other non-volatile storage medium). Or, the event can be
31 transmitted to another, fully independent event-processing apparatus.

1 Event durability permits re-processing the event, both in connection with
2 event processing and to facilitate recovery in case of data loss. The event ingestor **12**
3 undertakes no event processing until the event has been made durable.

4 **Delayed events and processing order**

5 In the course of the journey from an event source **16, 18, 20** to the event
6 ingestor **12**, an event can become delayed. This may be for a variety of reasons, such
7 as network congestion, propagation delay, or retransmission of event data due to
8 corruption or packet loss. As a result of event delays, events do not necessarily reach
9 the event ingestor **12** in an order that is consistent with the order in which the events
10 actually occurred (e.g., according to the order of their surrogate timestamps).

11 **Data Structures**

12 The event ingestor **12** features storage **22** in which is stored an event list **24**, a
13 set of first and second variables **26A, 26B** (generally **26**), and a set of corresponding
14 first and second chronologies **28A, 28B** (generally **28**). Although two variables and
15 two chronologies are shown, this is for illustration only. The method described herein
16 is applicable to one or more variables and one or more corresponding chronologies.

17 The values of the variables **26** define a working state of the event ingestor **12**.
18 An event-handler **30** enables the event ingestor **12** to change the working state as
19 events arrive. For example, the event-handler **30** may use a rule processing engine to
20 evaluate a set of rules based on incoming event data and trigger certain rules in
21 response to certain rule conditions being met. The triggered rules may then perform a
22 corresponding action.

23 So, for each incoming event, the event-handler **30** performs an action, which
24 may involve operations on the variables **26**. These actions are tied to whatever point
25 in time corresponds to the event (according to its surrogate timestamp). However,
26 because the actions are carried out in response to an event's arrival, and because the
27 events do not necessarily arrive in the correct order, the actions will also not be
28 executed in the correct order. In fact, it would be quite common for the actions to be
29 executed in an order that is inconsistent with the temporal order of the underlying
30 events with which they are associated. As a result, there will be times in which the

1 states for each variable are inconsistent with what they would have been had the
2 events arrived in sequence.

3 Each chronology **28** is associated with a corresponding variable **26**. A
4 chronology **28** includes a record of a set of events that affect that variable, as well as
5 the actions that result from the occurrence of that event. These actions include rule
6 invocations, and state changes in the form of specific operations performed on the
7 variable **26** corresponding to the chronology **28**.

8 As events arrive, the event-handler **30** incrementally builds the chronology **28**
9 by updating the chronology **28** for each new event received, for example, by adding a
10 new frame, as described in more detail below. However, the chronology **28** is not
11 immutable. When a delayed event arrives, and is processed out of order, the
12 chronology **28** can be revised.

13 **Anatomy of an event**

14 As shown in FIG. 2, each event has a surrogate timestamp **64** and a body **66**.
15 The surrogate timestamp **64** ensures that the event sequence is fully ordered. The
16 body **66** specifies what tasks are to be carried out in connection with the event. In
17 addition, each event also has a state field **68** and an accessed variables field **70**.

18 The event's state field **68** indicates whether the event is pending, valid, or
19 invalid. Every event starts out as pending. It stays pending until all tasks associated
20 with the event are completed. Then, the event becomes valid. A valid event may
21 become invalid if a delayed event that precedes it in time arrives. When an event
22 becomes invalid, an event-handler **30** must be spawned to re-process it.

23 The event's accessed-variables field **70** lists all the variables that were
24 accessed when that event was processed. It is initially blank and is updated after an
25 event has been processed or reprocessed.

26 Upon receiving an event, the event ingestor **12** spawns an event-handler **30** for
27 that event. The event-handler creates an entry in the event list **24** and marks it
28 "pending." Then, the event-handler **30** accesses one or more variables using an
29 operation determined by the event-handler **30** (e.g., an operation that reads, writes, or
30 both reads and writes the value of a variable **26**). As it does so, it accumulates a list of

1 the variables that were accessed, and, when it terminates, stores this list in the
2 accessed variables field **70**.

3 Upon completion, the event-handler **30** inspects the state field **68** for the event.
4 If the state field **68** is still set to “pending,” the event-handler **30** sets it to “valid.”
5 However, it is possible that while the event-handler **30** was busy carrying out its event
6 processing, a delayed event arrived. This means that the event ingestor **12** spawned an
7 event-handler for that delayed event, which might then have set the state field from
8 “pending” to “invalid.” If this happens, the event-handler **30** sets it back to “pending”
9 and reprocesses the event.

10 When an event-handler **30** reprocesses an event, it first receives a list of all
11 accessed variables. As it accesses variables during reprocessing, it uses the list of
12 accessed variables to identify those variables that are being revisited. When it revisits
13 an accessed variable, it marks it as valid. Since the event-handler knows all the
14 variables, and knows all those that have been accessed, it can easily determine which
15 variables were not accessed. It then proceeds to delete frames associated with those
16 variables, as described in more detail below.

17 **Anatomy of a chronology**

18 Chronologies are represented as a sequence of time-keyed frames **54** (FIG. 3),
19 each of which corresponds to an event. FIG. 3 shows a typical chronology in both an
20 abbreviated format **50** and an expanded format **52**. The expanded format **52** explicitly
21 shows each field of the frame **54**. The abbreviated format **50** omits certain fields, but
22 incorporates the information in those fields by judicious use of strikethroughs and
23 asterisks. For compactness of representation, this disclosure will often make use of the
24 abbreviated format **50**, which also identifies in the value field header the name of the
25 variable for which the chronology lists the sequence of operations and resulting values
26 for that variable.

27 The illustrated chronology **52** shows five frames, each of which has four
28 fields. In response to receiving an event, the event-handler **30** adds a frame **54**. This
29 new frame **54** has a time field **56**, a type field **58**, a value field **60**, and a validity field
30 **62**.

1 The time field **56** stores a surrogate timestamp. In the abbreviated format, a
2 blank indicates start of time. The “downstream” direction in a chronology is in the
3 direction of increasing values of the timestamp. The “upstream” direction is the
4 opposite of the downstream direction.

5 The type field **58** indicates the type of frame, which corresponds to the type of
6 operation that is performed on the variable in response to the event associated with
7 that frame. There are fundamentally two kinds of frames: read frames and write
8 frames. The write frames subdivide into two more types: a blind-write frame, and a
9 read-write frame.

10 A “read” frame is added when, as a result of a received event, the value of the
11 variable corresponding to the chronology is read, but no writing takes place. The
12 illustrated chronology shows two read frames, one at $t=20$ and another at $t=40$. In the
13 abbreviated format, an asterisk indicates a read frame.

14 A “read-write” frame is added when, as a result of a received event, a value of
15 the variable corresponding to the chronology has been both read and a new value of
16 that variable has been written into the chronology. In general, the value written
17 depends on the value that was read. The illustrated chronology has two read-write
18 frames, one at $t=30$ and another at $t=50$. These are indicated in the abbreviated format
19 by the written value with no bar above the value. For compactness of representation,
20 the expanded format refers to a read-write frame as an “update.”

21 A “blind-write” frame is added when, as a result of a received event, a value in
22 the chronology has been overwritten, but without having been read. The illustrated
23 chronology has only two blind-writes. One is upon initialization at $t=0$ and the other is
24 at $t=10$. In the abbreviated format, a blind-write is indicated by a solid line above the
25 value that was blindly written. For compactness of representation, in the expanded
26 format, a blind-write is referred to as an “overwrite.”

27 The distinctions between a read, a read-write, and a blind-write are important
28 because they will eventually be used to determine whether the event that spawned the
29 frame will need to be reprocessed in response to receiving a delayed event.

1 The value field **60** includes the newest value of the variable as of the time that
2 the frame **54** was added. This value field **60** is only used when the type field **58**
3 indicates that the frame **54** is a blind-write frame or a read-write frame.

4 Finally, the validity field **62** indicates whether or not the frame **54** is valid. A
5 frame starts out as valid. But as new events arrive, it can be rendered invalid. In the
6 abbreviated format, valid and invalid values are indicated by the absence and presence
7 of a strike-through.

8 In response to an event, the event-handler **30** updates a chronology (which is
9 different from the “updates” performed on the variables) by performing operations on
10 a chronology involving its frames (which are different from the “operations”
11 performed on the variables). In general, there are four kinds of operations on a
12 chronology: an “append,” an “insert,” a “modify,” and a “delete.”

13 An “insert” operation adds a frame before the last frame of a chronology. An
14 “append” operation is like an insert, except it inserts the frame after the last frame of a
15 chronology. A “modify” changes something in the frame. And a “delete” removes a
16 frame from a chronology altogether.

17 **Significant and insignificant operations**

18 A particular operation to update a chronology **28** is either “significant” or
19 “insignificant.” A “significant” operation will result in a revision, or change, to a
20 value that is visible at a later point in a chronology. More specifically, a “significant”
21 operation will affect the outcome of actions that were temporally after the delayed
22 event, with the result that consequent state changes resulting from those actions may
23 need to be rescinded.

24 FIGS. 4A-4B illustrate some examples of how one determines if an insertion
25 of a frame in a chronology is a significant insertion or an insignificant insertion. In
26 general, an insert operation is significant if it calls into question values that are
27 located downstream from it.

28 In the operation **110** shown in FIG. 4A, an event has caused an append
29 operation that results in an append frame with the value \$300 at $t=35$. This operation
30 is not significant because there are no downstream values after $t=35$ that could

1 possibly be affected. In general, an append operation cannot be significant (i.e., will
2 always be insignificant), because it is by definition after the last frame.

3 In operation **112**, a delayed event has caused insertion of a read frame at $t=8$.
4 This operation is not significant because a read cannot affect values in the chronology.
5 In general, a read operation will always be insignificant, because it does not change
6 the value being read.

7 A blind-write or a read-write can sometimes be significant because it can
8 sometimes cause subsequent frames to be invalid.

9 In operation **114**, a delayed event at $t=17$ has caused a read-write operation
10 resulting in insertion of a read-write frame. At the time the downstream read frame at
11 $t=20$ was generated, the delayed event would not yet have arrived. Therefore, the
12 event-processor would have read the value \$2.25. But as a result of the delayed event
13 at $t=17$, this would now be incorrect. As a result, the downstream read frame at $t=20$
14 must be invalidated. This invalidity will later be corrected by reprocessing the event at
15 $t=20$.

16 Similarly, the downstream read-write frame at $t=30$ is invalidated because it
17 involved a read operation. Because the delayed event would not yet have arrived at
18 the time this read-write frame was added, the read-write frame is based on having
19 read a value that is now known to be incorrect. As a result, it too must be marked
20 invalid.

21 In general, a read-write frame will invalidate all downstream frames that
22 involve a read, whether as a result of a read operation or a read-write operation.
23 However, a blind-write operation, which requires no read, will not be invalidated.

24 Not all read-write operations will trigger invalidity propagation. In operation
25 **116**, a delayed event has caused insertion of a read-write frame at $t=9$ just before a
26 blind-write frame at $t=10$. Yet, this does not invalidate the blind-write frame. This is
27 because a blind-write frame does not involve reading anything. Therefore, the value
28 written in a blind-write could not possibly be affected by upstream values in the
29 chronology. Any frame downstream from a blind-write that involves a read would
30 also not be invalid because such a read would read the value in the blind-write frame.

1 Thus, a blind-write frame acts as a kind of barrier frame that blocks downstream
2 invalidity propagation.

3 Insertion of a read-write operator thus causes invalidity to propagate
4 downstream from its point of insertion. This invalidity propagates until either the end
5 of the chronology or until a barrier frame, such as a blind-write frame is reached. The
6 barrier frame thus operates as a barrier that stops invalidity from propagating past it.
7 This limits the extent of reprocessing required in response to a delayed event. Instead
8 of having to reprocess all events that come after insertion of a read-write frame, only
9 those events that come before a blind-write will need to be reprocessed.

10 A read-write frame does not always invalidate downstream frames that involve
11 a read. For example, in the configuration **118**, a delayed event has caused insertion of
12 a read-write frame at $t=13$ directly after the blind-write at $t=10$. But this read-write
13 frame does not trigger downstream invalidity propagation because the value in the
14 read-write frame at $t=13$ is the same as that in the blind-write frame at $t=10$. Thus, the
15 read-write frame at $t=13$ really makes no difference. Values read by any downstream
16 read operations or read-write operations would not have changed as a result of the
17 insertion of a read-write frame at $t=13$.

18 FIGS. 4A-4B also illustrate some examples of how one determines if
19 modification of a frame in a chronology is a significant or insignificant. Like the
20 insert operation, the modify operation sometimes, but not always, renders downstream
21 frames invalid.

22 In the operation **120**, a delayed event has caused the frame at $t=15$ to be
23 modified to have a new value, \$2.30. This starts downstream invalidity propagation.
24 The invalidity propagation continues until it reaches either a barrier frame or the end
25 of the chronology. The reason it does so is the same as that discussed in connection
26 with the read-write frame.

27 On the other hand, in the operation **122**, a delayed event has caused the frame
28 at $t=15$ to be modified. But the modified value of the frame at $t=15$ is the same as its
29 original value. This obviously cannot affect anything downstream. Therefore, this
30 modification is insignificant.

1 FIGS. 4A-4B also illustrate some examples of how one determines if deletion
2 of a frame from a chronology is significant or insignificant. Like the insert operation
3 and the modify operation, a deletion operation sometimes, but not always, renders
4 downstream frames invalid.

5 First of all, deleting a read frame, as shown in the operation **124**, is never
6 significant. After all, if inserting a read frame made no difference, it would seem odd
7 for deleting the same read frame to make any difference.

8 On the other hand, deleting a blind-write frame or a read-write frame will be
9 significant if it changed a value, as shown in the operation **126**, but not if it made no
10 change to a value, as shown in the operation **128**.

11 Operation **130** illustrates a blind-write frame acting as a barrier frame that
12 prevents invalidity from propagating downstream. In the example shown, a delayed
13 event has caused a modification to the frame at $t=10$. Ordinarily, this modification
14 would be significant for reasons discussed above in connection with operation **120**.
15 However, the immediately following frame at $t=15$ is a blind-write frame. This
16 operates to immediately block invalidity propagation downstream. Thus, no frames
17 are rendered invalid.

18 In some embodiments, not all potentially invalid frames are marked. This is
19 because in some cases, whether or not a frame is invalid is a function of the correct
20 value of one or more invalid frames.

21 For example, in operation **132**, a read-write frame is inserted at $t=15$. As
22 expected, the read at $t=20$ is marked invalid. Since the read-write frame at $t=30$
23 depends on what is read at $t=20$, it too is marked invalid. Since the read frame at $t=35$
24 is also downstream from the significant insertion at $t=15$, one might expect it to also
25 be marked invalid. And in fact, in one embodiment, this would be the case.

26 However, it is also possible that when the event at $t=30$ is ultimately
27 reprocessed, the value at $t=30$ will ultimately be replaced by \$2.75. This means it will
28 not have changed. In that case, the read frame $t=35$ would not be invalid.

1 Because there is a possibility that the read frame at $t=35$ will ultimately be
2 valid, it remains marked as valid. Whether or not it truly is valid will not be known
3 until the event at $t=30$ has been reprocessed. If, upon reprocessing the event at $t=30$, it
4 is found that the read frame at $t=35$ has been rendered invalid, then and only then will
5 the read frame at $t=35$ be marked invalid.

6 The foregoing method, in which the validity of a valid frame remains
7 unresolved in the face of a significant operation upstream from it, reduces
8 computational load by avoiding unnecessary event re-processing. Such a frame thus
9 also a barrier frame in the same way that a blind-frame is a barrier frame. The
10 difference between the two is that a frame of unresolved validity forms a conditional
11 barrier, whereas a barrier formed by a blind-write frame forms an unconditional
12 barrier.

13 The operation **134** is similar to operation **132** with the exception that the frame
14 at $t=30$ is a blind-write frame instead of a read-write frame. In this case, the frame at
15 $t=30$ is an unconditional barrier frame, rather than a conditional barrier frame as was
16 the case in operation **132**.

17 In the method described herein, as computation progresses, the probability of
18 rescinding a state change in a chronology **28** approaches zero. As a result, a
19 chronology **28** converges to its deterministic final state.

20 **Example 1: Cumulative maximum with no event delays**

21 FIG. 5 illustrates steps in execution of the method in the absence of event
22 delays for an event list **32**. The illustrated process concerns tracking the maximum
23 value attained by of each of two variables “AAA” and “BBB.” These variables might
24 represent, for example, stock prices. These

25 The event list **32F** shows six events, each of which corresponds to a value for
26 one of the two variables.

27 Upon receiving the first event, the event-handler **30** updates the event list **32A**
28 to indicate the value of the first variable. The event-handler **30** also initializes a first
29 chronology **34A** for the first variable and enters the appropriate value.

1 Upon receiving the second event, the event-handler **30** updates the event list
2 **32B** to indicate the value of the second variable. The event-handler **30** also initializes
3 a second chronology **36B** for the second variable and enters the appropriate value.

4 Upon receiving the third event, the event-handler **30** updates the event list
5 **32C**. Since the event concerns the second variable, the first chronology **34C** is
6 unchanged. The event-handler **30** peeks at the second chronology **36C**. This act of
7 peeking is recorded in the second chronology **36C**. An asterisk in the illustration of
8 the second chronology **36C** represents the recordation of this act of peeking.
9 However, since the value received in the third event is lower than the value already in
10 the second chronology **36C**, no change has to be made to the second chronology **36C**.

11 Upon receiving the fourth event, the event-handler **30** updates the event list
12 **32D**. Since the event concerns the second variable, the first chronology **34D** is
13 unchanged. The event-handler **30** peeks at the second chronology **36D** and discovers
14 that this time, the maximum value of the second variable has changed. As a result, it
15 updates the second chronology **36D**.

16 Upon receiving the fifth event, the event-handler **30** updates the event list **32E**.
17 Since the event concerns the first variable, the second chronology **36E** is unchanged.
18 The event-handler **30** peeks at the first chronology **34E** and discovers that the
19 maximum value of the first variable has changed. Therefore, the event-handler updates
20 the first chronology **34E**.

21 Upon receiving the sixth event, the event-handler **30** updates the event list
22 **32F**. Since the event concerns the first variable, the second chronology **36F** is
23 unchanged. The event-handler **30** peeks at the first chronology **34F**. This act of
24 peeking is recorded in the first chronology **34F**. An asterisk in the illustration of the
25 first chronology **34F** represents the recordation of this act of peeking. However, since
26 the value received in the sixth event is lower than the value already in the first
27 chronology **34F**, no change has to be made to the first chronology **34F**.

28 As is apparent, the first and second chronologies **34F**, **36F** have converged to
29 the state that one would expect given the complete list of events **36F**.

1 **Example 2: Cumulative maximum with event delays**

2 FIG. 6 considers the case in which the events shown in the event list **36F**
3 arrive in the wrong order.

4 Upon receiving the first event, the event-handler **30** updates the event list **38A**
5 to indicate the value of the first variable. The event-handler **30** also initializes a first
6 chronology **40A** for the first variable and enters the appropriate value.

7 Upon receiving the second event, the event-handler **30** updates the event list
8 **38B** to indicate the value of the second variable. The event-handler **30** also initializes
9 a second chronology **42B** for the second variable and enters the appropriate value.

10 However, as a result of event delays, the next event to arrive, however, is not
11 the third event but the fourth event. Upon receiving the fourth event, the event-
12 handler **30** updates the event list **38C** to indicate the value of the second variable. The
13 event-handler **30** also updates the second chronology **42C**. Since the event does not
14 involve the first variable, the event-handler **30** leaves the first chronology **40C** alone.

15 Then, the third event finally arrives. Upon receiving it, the event-handler **30**
16 updates the event list **38D** by inserting the third event in the correct position. Since the
17 third event concerns only the second variable, the first chronology **40D** can be left
18 alone.

19 The event-handler **30** peeks at the second chronology **42D**. The event-handler
20 **30** then updates the second chronology **42D** by recording this peek, as indicated by
21 the asterisk. Based on the nature of the update caused by the operation, the event-
22 handler **30** must now classify this operation as significant or insignificant.

23 To do so, the event-handler observes, from having peeked at the second
24 chronology **42D**, that the value of the second variable in the third event is not high
25 enough to change anything. As a result, the operation is deemed insignificant and the
26 event-handler **30** also leaves the value of the second variable in the second
27 chronology **42D** alone.

28 Again, as a result of event delays, the next event to arrive is the sixth event,
29 not the fifth one. The event-handler updates the event list **38E** accordingly, updates

1 the first chronology **40E**, and leaves the second chronology **42E** alone. However, in
2 reality, the first chronology is now incorrect because a higher value is reflected in the
3 delayed fifth event.

4 Finally, the fifth event arrives. The event-handler **30** updates the event list **38F**
5 by inserting the fifth event in the correct location. Since the fifth event does not affect
6 the second chronology **42F**, the event-handler **30** concerns itself with only the first
7 chronology **40F**.

8 In particular, the event-handler **30** updates the first chronology by inserting the
9 fifth event in the correct location in the first chronology. It then checks to see if the
10 update caused by the operation associated with this fifth event is sufficient to classify
11 the operation as significant.

12 In doing so, the event-handler **30** peeks at the values for events after the fifth
13 event in the first chronology **40F**. It discovers that the event-handler that handled the
14 sixth event relied on a value that has been rendered incorrect by the update. Thus,
15 based on the update it caused, this operation on the first chronology **40F** is deemed
16 significant.

17 In response to determining that an operation is significant, the event-handler
18 **30** for the fifth event marks the actions carried out by the event-handler **30** for the
19 sixth event as invalid. This is shown by the entries that have been stricken-through in
20 the first chronology **40F** and in the event list **38F**.

21 Because a significant operation occurred, the event-handler re-runs the event-
22 handlers for all affected events. In this case, the only affected event is the sixth event.

23 The event-handler **30** for the sixth event then processes the sixth event once
24 again. In doing so, it writes over the entry of the sixth event in the event list **38G** and
25 peeks at the first chronology **40G**. An asterisk in the first chronology **40G** indicates
26 that the peek has been recorded. The event-handler **30** then recognizes that no update
27 to the first chronology **40G** is necessary since the value of the sixth event is less than
28 the current maximum value.

1 It is apparent that the chronologies in FIG. 5 and FIG. 6 managed to converge
2 to the same final values. However, the intermediate states of the chronologies in FIG.
3 6 included at least one incorrect state, whereas in FIG. 5, the chronologies are always
4 in the correct state. Thus, the paths followed from the initial state to the final state in
5 FIGS. 5 and 6 were different.

6 Upon arrival of an event, the event processing system spawns an event-handler
7 **30** to process the event. The event-handler accesses the variables and the chronologies
8 of those variables. These variables are shared and can be accessed by other event
9 ingestors **12**. As it does so, it builds frames and other data structures that are used in
10 connection with determining what events need to be reprocessed. Then, the event-
11 handler **30** terminates.

12 **Example 3: Clearing payments: no event delays**

13 FIG. 7 illustrates another example in which there are three accounts, *X*, *Y*, and
14 *Z*, each associated with a bank customer, and a fourth account, *OD*, that collects fees
15 for overdrafts. In this example, if a customer's balance is overdrawn by less than a
16 defined cap, a \$10 overdraft fee is debited from that customer's account and credited
17 to the bank's overdraft account.

18 This scenario is one that is particularly ripe for the sort of trouble that can be
19 caused by a delayed event. For instance, it is quite common for one to replenish a
20 depleted account by depositing a large check, and to then immediately start paying
21 bills. This began a race. If the bills were to win the race, i.e., if the bills are processed
22 before the check has cleared, there would be a flurry of overdraft charges.

23 In the past, this rarely happened because bills were often paid by mail,
24 effectively handicapping one of the racers. However, in modern transaction
25 processing, many bills are paid electronically, effectively removing this built-in
26 handicap.

27 In FIG. 7, there are no delayed events. The process begins with transferring
28 money from account *Z* to account *X* (step **42A**), thus raising *X*'s balance from \$10 to
29 \$20. As a result, when *X* pays off his \$15 debt to *Y* (step **42B**), *X*'s \$20 balance is
30 sufficient, and no overdraft occurs.

1 Example 4: Clearing payments: event delays

2 FIG. 7 starts with the same initial state **44A**. However, the event
3 communicating the transfer from account *Z* to account *X* is delayed (step **44B**). As a
4 result, when the \$15 transfer from account *X* to account *Y* is processed, account *X* has
5 only \$10. Because of this, an overdraft fee is credited to account OD (step **44C**).

6 The delayed event finally arrives (step **44D**). The first step is to update
7 account *Z* and *X* by debiting the former \$10 (step **44E**) and crediting the latter \$10
8 (step **44F**). The update to account *X* renders the earlier event invalid. Note that the
9 value in the *Y* account is not invalidated.

10 As a result of having been rendered invalid, the earlier event must be
11 reprocessed. This results in correcting the value for account *X* (step **44G**) and deletion
12 of the overdraft in account OD (step **44H**).

13 Once again, the four chronologies have converged into the correct state.

14 Example 5: Asynchronous processing

15 In the course of processing an event, it may be necessary to update several
16 variables. It is advantageous, at least in conventional methods, for an event processing
17 operation to be atomic. This generally requires that obtaining latches (or “locks”) on
18 all variables to be updated. This is a temporally expensive operation. In addition, it
19 creates the possibility of deadlock.

20 The method described herein dispenses with the need for atomicity in an event
21 processing operation. Instead of obtaining multiple latches, variables are updated in a
22 globally asynchronous manner. At times, the resulting configuration of variable
23 values may be incorrect. However, the nature of the method is such that, in the long
24 run, the configuration of variable values will converge to a correct state. A latch on
25 the chronology may be may be obtained to enable atomic an atomic set of operations
26 to the chronology (e.g., inserting and deleting frames), such brief local latches on a
27 chronology are less expensive than latches on the variables themselves, which may be
28 shared among multiple nodes.

29 As shown in FIG. 8, a first event **72A** and a second event **72B** arrive at almost
30 the same time, as a result of which first and second event-handlers are executing

1 concurrently. The first event **72A** requires that \$10 be transferred from account *Z* to
 2 account *X*. This requires two operations: a debit to account *Z* and a credit to account
 3 *X*.

4 The first event handler manages to perform the debit from account *Z* (step **74**)
 5 by updating a chronology **88** for account *Z*. However, before it has a chance to carry
 6 out the corresponding credit to account *X*, the second event handler steps in to process
 7 the second event **72B**. This results in an update to the chronology **84** for account *X*
 8 that shows a debit \$15 from account *X*. This triggers an overdraft in account *X* (step
 9 **76**). It also results in an apparent disappearance of \$10 that was supposed to have
 10 been moved into account *X*.

11 The second event-handler, which is completely oblivious to the fact that it
 12 interrupted the first event-handler, completes processing the second event **72B** by
 13 updating a chronology **90** for account *OD*, thus crediting the bank's overdraft account
 14 (step **78**).

15 Finally, the first event-handler finishes processing the first event **72A**. This
 16 restores the missing \$10 into account *X* and also invalidates the processing carried out
 17 by the second event-handler (step **80**).

18 Invalidation of the second event **72B** triggers the spawning of an event-
 19 handler to re-process the second event **72B**. This results in correction of the balance,
 20 and deletion of a frame from the overdraft chronology (step **82**). As a result, the
 21 chronologies **84**, **86**, **88**, **90** have converged to the correct states without the use of
 22 latches.

23 **Example 6: Chronology Reconstruction**

24 Due to potentially frequent events, and their associated operations,
 25 chronologies are typically stored in relatively fast storage (e.g., memory or other
 26 volatile storage). However, this leaves chronologies vulnerable to data loss in the
 27 event of hardware failure. The method described herein provides a simple way to
 28 reconstruct all or part of a chronology that may have been lost as a result of a
 29 hardware failure. To enable this feature, the event ingestor **12** writes a checkpoint to
 30 durable storage (e.g., a disk or other non-volatile storage). The checkpoint identifies a
 31 value of a variable corresponding to the chronology and the most recent event during

1 which the value of that variable was valid. Armed with this checkpoint and an event
2 list it is possible to reconstruct the complete chronology. The event list itself is
3 assumed to be much less expensive to maintain in durable storage because events are
4 not revised, and therefore each event only needs to be stored once.

5 FIG. 10 begins with a chronology **92** that is to be recovered (step **94**). As a
6 result of some failure, a portion of the chronology **92** has been lost (step **96**). All that
7 is left is a checkpoint value **98**.

8 The first step in recovery is to identify those events that need to be processed.
9 This results, in this case, in invalidation of first and second events **102, 104** (step
10 **100**).

11 As a result of invalidation, first and second event-handlers are spawned to
12 reprocess the corresponding first and second events **102, 104**. Thus results in
13 reprocessing of the first event **102** (step **106**) and reprocessing of the second event
14 **104** (step **108**).

15 The reconstruction method does not require reprocessing of all events, but
16 only those events that are required to reconstruct the lost chronology **94**.

17 **Example 7: Partial Chronology Reconstruction**

18 In some cases, a chronology will have been damaged in a way that makes it
19 difficult to determine if an event has been properly processed. In that case, one cannot
20 tell if the event should be reprocessed or not. A conservative option is to reprocess all
21 events, as described in the preceding example. However, this is computationally
22 wasteful.

23 In an alternative embodiment, each frame further includes a generation count
24 associated with each data access to the chronology that contains that frame. In this
25 embodiment, every chronology access is accompanied by changing the generation
26 count of the frame that was accessed. The combination of a particular frame's
27 generation count and the surrogate timestamp of the event that led to accessing that
28 frame defines a unique access tuple. When processing an event, an event processor
29 begins by storing a starting and ending generation count. These can be used to later to
30 determine whether the event must be reprocessed.

1 The process of using generation counts is illustrated in FIG. 11.

2 In the step **136** shown in FIG. 11, the event to be processed is the initializing
3 of four chronologies. Processing this event required accessing each of the four
4 chronologies. Each access is associated with a generation count. As shown in the
5 figure, each chronology has a unique generator count, g , associated with that event.
6 The initialization step began with a value of 1 for the generator count and ended with
7 a value of 4 for the generation count.

8 In step **138**, the event to be processed is the transfer of \$15 from the X
9 chronology to the Y chronology. Processing begins with the event frame having a
10 starting generation count $s=1$. This is the first value of generation count that will be
11 assigned during processing this event. The ending generation count, e , is left blank
12 until the event is fully processed.

13 The next step in processing the event is to append a read-insert frame to the X
14 chronology to reflect a deduction of \$15 and a \$10 overdraft fee (step **139**). This
15 requires a data access to the X chronology. To indicate this data access, the event
16 processor marks the X chronology with a generation count $g=1$.

17 Event processing continues with step **140**, in which a read-write frame is
18 appended to the Y chronology. This requires access to the Y chronology. This access is
19 the second access made during processing of the event. As a result, that read-write
20 frame is tagged with a generation count of $g=2$.

21 Event processing requires access to one more chronology, the OD chronology.
22 The next step, shown in step **144**, is to append a read-write frame to the OD
23 chronology. Since this would be the third chronology access made in the course of
24 processing the particular event, this appended read-write frame is tagged with a
25 generation count of $g=3$.

26 At this point, processing the event is complete, and all that remains is to
27 update the ending generation count, e , for this event to "3."

28 This state of completion does not last long, however. In step **146**, a delayed
29 event at $t=1$ arrives. This event includes instruction to transfer \$10 from the Z account

1 to the X account. Had this event arrived earlier, i.e. before the event at $t=2$, the
2 overdraft would never have occurred. Now that it has arrived, the overdraft must be
3 undone and everything else put to right.

4 As shown in step **146**, the event at $t=1$ is shown at the end of event processing.
5 Processing the event at $t=1$ required two accesses. This results in starting and ending
6 generation counts of $s=1$ and $e=2$ respectively in the event frame for $t=1$. As a result
7 of the late arrival of the event at $t=1$, the event at $t=2$ is now invalid. Its state is now
8 marked “pending” to show that reprocessing of the event at $t=2$ has begun.

9 In anticipation of another access, the starting generator count for the event
10 frame of the event at $t=2$ is now incremented to $s=4$. Its ending generation count, e , is
11 deleted since it will not be known until completion of event reprocessing.

12 In step **148**, reprocessing of the event at $t=2$ is complete. The end generation
13 counter for the event is filled in. And the chronologies have all converged to a state
14 that is consistent with all known events.

15 FIG. 12 inventories all the chronology states that the chronologies in FIG. 11
16 passed through. Each column in FIG. 12 represents one node. FIG. 13 does the same
17 for all the event frames. Each column in FIG. 13 corresponds to one event.

18 It is apparent from the examples in FIGS. 11 and 12 that there are several
19 conditions that must be present for an event to be valid. First, the event frame should
20 be marked valid. Second, the access field of the event frame should match the
21 accesses in the chronologies. Third, the value frames are all marked valid. And last,
22 the generation counts in the chronologies should match the range of generation counts
23 specified in the event frame. If any one of these conditions is not met, the event
24 cannot be valid.

25 It is assumed that there exists a decentralized logging system and that any
26 changes to chronologies or to the event list are saved to logs such that after a failure, it
27 is possible to recover chronologies and the event list to some earlier state. Because
28 logging is decentralized, and because not all chronologies are stored in the same node,
29 it is possible to have, after a node failure, any combination of states in FIG. 11. The
30 idea is to determine which events, if any need to be reprocessed. Using generation

1 counters as described below, it is possible to determine whether the combination of
2 events is valid. This avoids the need to reprocess events unnecessarily.

3 FIG. 14 shows examples of how to use the event frames, and the above four
4 conditions, to determine whether the chronologies are correct.

5 In example **150**, all event frames are marked as valid. But in fact, on closer
6 inspection, particularly inspection of generation counters, it is apparent that the event
7 at $t=2$ is invalid and requires reprocessing. In particular, the starting and ending
8 generation counters of the event at $t=2$ define a closed interval $[4, 5]$. Thus, the
9 chronologies should show generation counters in the interval $[4, 5]$. But they do not.
10 The X chronology has a generator counter of 1, which is outside the interval.

11 In example **152**, although all events are marked as valid, the event at $t=1$ must
12 be invalid. This is because the event frame specifies that the chronologies for the
13 event should have generation counts in the interval $[1, 2]$. In reality, the only
14 generation count is $g=1$. The X chronology has apparently not yet been accessed. The
15 event at $t=2$ is also invalid because the event list requires that all generation counts in
16 the corresponding chronologies lie in the interval $[4, 5]$. However, the generation
17 counts actually line in the interval $[1, 5]$. Thus, both events require reprocessing.

18 It is possible to arrive at the same result by inspecting the access field of the
19 event frame. According to the access field of the event frame for $t=1$, there should
20 have been an access to both the Z chronology and the X chronology. According to the
21 chronologies themselves, only the Z chronology was accessed.

22 In example **154**, both events at $t=1$ and $t=2$ are invalid. This is because the
23 access field in the event frame for the event at $t=1$ shows that there should have been
24 an access to both the X chronology and the Z chronology. As is apparent, nothing has
25 accessed the X chronology. The same reasoning applies to the event at $t=2$.

26 In example **156**, the event at $t=1$ is invalid because, according to the event
27 frame at $t=1$, the generation counts should all be in the interval $[1, 2]$. However, in the
28 corresponding chronologies, the generation counts are in the degenerate range $[1, 1]$.
29 The event at $t=2$ is also invalid because the predicted generation count range in the
30 event frame, $[4, 5]$, does not match what is in the chronologies.

1 The event-processing approach described above can be implemented, for
2 example, using a programmable computing system executing suitable software
3 instructions or it can be implemented in suitable hardware such as a field-
4 programmable gate array (FPGA) or in some hybrid form. For example, in a
5 programmed approach the software may include procedures in one or more computer
6 programs that execute on one or more programmed or programmable computing
7 system (which may be of various architectures such as distributed, client/server, or
8 grid) each including at least one processor, at least one data storage system (including
9 volatile and/or non-volatile memory and/or storage elements), at least one user
10 interface (for receiving input using at least one input device or port, and for providing
11 output using at least one output device or port). The software may include one or
12 more modules of a larger program, for example, that provides services related to the
13 design, configuration, and execution of dataflow graphs. The modules of the program
14 (e.g., elements of a dataflow graph) can be implemented as data structures or other
15 organized data conforming to a data model stored in a data repository.

16 The software may be stored in non-transitory form, such as being embodied in
17 a volatile or non-volatile storage medium, or any other non-transitory medium, using
18 a physical property of the medium (e.g., surface pits and lands, magnetic domains, or
19 electrical charge) for a period of time (e.g., the time between refresh periods of a
20 dynamic memory device such as a dynamic RAM). In preparation for loading the
21 instructions, the software may be provided on a tangible, non-transitory medium, such
22 as a CD-ROM or other computer-readable medium (e.g., readable by a general or
23 special purpose computing system or device), or may be delivered (e.g., encoded in a
24 propagated signal) over a communication medium of a network to a tangible, non-
25 transitory medium of a computing system where it is executed. Some or all of the
26 processing may be performed on a special purpose computer, or using special-purpose
27 hardware, such as coprocessors or field-programmable gate arrays (FPGAs) or
28 dedicated, application-specific integrated circuits (ASICs). The processing may be
29 implemented in a distributed manner in which different parts of the computation
30 specified by the software are performed by different computing elements. Each such
31 computer program is preferably stored on or downloaded to a computer-readable
32 storage medium (e.g., solid state memory or media, or magnetic or optical media) of a
33 storage device accessible by a general or special purpose programmable computer, for

1 configuring and operating the computer when the storage device medium is read by
2 the computer to perform the processing described herein. The inventive system may
3 also be considered to be implemented as a tangible, non-transitory medium,
4 configured with a computer program, where the medium so configured causes a
5 computer to operate in a specific and predefined manner to perform one or more of
6 the processing steps described herein.

7 A number of embodiments of the invention have been described. Nevertheless,
8 it is to be understood that the foregoing description is intended to illustrate and not to
9 limit the scope of the invention, which is defined by the scope of the following
10 claims. Accordingly, other embodiments are also within the scope of the following
11 claims. For example, various modifications may be made without departing from the
12 scope of the invention. Additionally, some of the steps described above may be order
13 independent, and thus can be performed in an order different from that described.

14 Having described the invention, and a preferred embodiment thereof, what is
15 claimed as new, and secured by Letters Patent is:

16

CLAIMS

1. A method for causing a computing system to process events from a sequence of events that defines a correct order for said events independent from an order in which those events are received over an input device or port, said method including:
 - defining a first variable,
 - defining, for said first variable, a first chronology of operations on said first variable associated with received events,
 - receiving a first event that pertains to said first variable,
 - executing a first operation on said first variable, wherein said first operation results in a first update of said first chronology,
 - after having received said first event, receiving a delayed event that pertains to said first variable,
 - executing a second operation on said first variable, wherein said second operation results in a second update of said first chronology, and
 - determining whether said first update is valid or invalid,
 - wherein said delayed event precedes said first event in said sequence,
 - wherein said first update is based on said first event, and
 - wherein said second update is based on said delayed event.
2. The method of claim 1, wherein determining whether said first update is valid or invalid includes determining that said first update is valid.

3. The method of claim 2, wherein determining that said first update is valid includes determining that said first update is the insertion of a write operation before said first update.
4. The method of claim 2, wherein determining that said first update is valid includes determining that said delayed event resulted in insertion of a read operation before said first update.
5. The method of claim 2, wherein determining that said first update is valid includes determining that said delayed event required insertion of an update operation before said first update.
6. The method of claim 2, wherein determining that said first update is valid includes determining that said first update is not affected by said second update.
7. The method of claim 1, wherein determining whether said first update is valid or invalid includes determining that said first update is affected by said second update, and rescinding said first update in response to determining that said first update is invalid.
8. The method of claim 1, wherein determining whether said first update is valid or invalid includes determining that said first update is affected by said second update, causing said first update to be invalid.
9. The method of claim 8, wherein determining that said first update is affected by said second update includes determining that said delayed

event required insertion of a chronology entry indicating an operation that includes writing said first variable before a chronology entry corresponding to said first update.

10. The method of claim 8, wherein determining that said first update is affected by said second update includes determining that said delayed event required updating an existing chronology entry indicating an operation on said first variable before a chronology entry corresponding to said first update.
11. The method of claim 1, wherein said correct order for said sequence of events is partially ordered, and said method further includes causing said sequence of events to be completely ordered.
12. The method of claim 1, wherein receiving said first event includes receiving said event from a set of completely ordered events.
13. The method of claim 1, further including
 - detecting a loss of data in said first chronology,
 - retrieving a first checkpoint value indicative of a valid value of said first variable in said first chronology,
 - based on said checkpoint value, retrieving a subset of events from a list of events, said list of events including said first event,
 - reprocessing said subset of events, thereby reconstructing said data lost in said loss of data, and
 - revising said first chronology to include said data lost in said

loss of data.

14. The method of claim **1**,

wherein said first chronology includes a plurality of entries corresponding to operations resulting from a plurality of events that are after an entry corresponding to a delayed event operation corresponding to said delayed event,

wherein executing a second operation includes identifying a barrier operation that blocks propagation of invalidity in said plurality of entries, and

revising only those operations from said plurality of operations whose entries are between entries corresponding to said delayed event operation and said barrier operation.

15. The method of claim **14**, wherein identifying a barrier operation includes identifying a write operation as a barrier operation.

16. The method of claim **1**, wherein determining whether said first update is valid includes

identifying an access to said first chronology that must have occurred to correctly execute said first update,

inspecting said first chronology, and

based on said inspection, determining whether said access has occurred.

17. The method of claim **16**, further including

maintaining first information and second information,

wherein said first information identifies an access to said first chronology, and

wherein said second information identifies a range of accesses required to process said first event, and

determining that said first information is within said range of accesses.

- 18.** The method of claim **1**, wherein the computing system includes a plurality of nodes in communication over a network, and said method further includes causing said computing system to execute the steps of

storing said first variable and said first chronology at a first node;

storing a second variable and a second chronology of operations on said second variable at a second node; and

processing at least some events of said sequence of events by performing operations on both said first and second variables, and updating both of said first and second chronologies in response to processing said events.

- 19.** Software, stored in non-transitory form on a computer-readable medium, for causing a computing system to process events from a sequence of events that defines a correct order for said events independent from an order in which those events are received over an input device or port, said software including instructions for causing a computing system to:

define a first variable,

define, for said first variable, a first chronology of operations on said first variable associated with received events,

receive a first event that pertains to said first variable,

execute a first operation on said first variable, wherein said

first operation results in a first update of said first chronology,
receive, after having received said first event, a delayed event
that pertains to said first variable,
execute a second operation on said first variable, wherein said
second operation results in a second update of said first
chronology, and
determine whether said first update is valid or invalid,
wherein said delayed event precedes said first event in said
sequence,
wherein said first update is based on said first event, and
wherein said second update is based on said delayed event.

- 20.** A computing system for processing events from a sequence of events that defines a correct order for said events independent from an order in which those events are received over an input device or port, the computing system including:

an input device or port configured to receive said events; and

at least one processor configured to participate in processing of said

sequence of events, said processing including

defining a first variable,

defining, for said first variable, a first chronology of
operations on said first variable associated with received
events,

receiving a first event that pertains to said first variable,

executing a first operation on said first variable, wherein said

first operation results in a first update of said first chronology,
after having received said first event, receiving a delayed
event that pertains to said first variable,
executing a second operation on said first variable, wherein
said second operation results in a second update of said first
chronology, and
determining whether said first update is valid or invalid,
wherein said delayed event precedes said first event in said
sequence,
wherein said first update is based on said first event, and
wherein said second update is based on said delayed event.