Neural, Parallel & Scientific Computations 15 (2007) 103-124

Quantitative Causality

Sharon Simmons and Dennis Edwards

Department of Computer Science, University of West Florida
Pensacola, FL 32514-5750

Abstract
Events generated by the execution of a distributed system are related by causality and concurrency. While providing a means of reasoning about the relative occurrence of events, this partial order fails to represent the timeliness of occurrence. In this paper, we develop a novel means of assigning weights to events where the weights are reduced as the temporal proximity to an anchor event decreases. This weight quantifies the strength of the causal or concurrent relationship with respect to an anchor event. Those events that causally succeed the anchor are the focus of this paper with concurrency and causally preceding being part of future work plans. Three methods of computing event weights for causally succeeding events are defined. Each contains a tunable parameter to determine the rate of weight decrease. The methods are piece-wise linear, exponential, and relevant vector difference decay. A case study has been performed that applied quantitative causality to the well-known software engineering problem of feature location. A summary of the case study results is provided to illustrate the utility of quantitative causality for succeeding events.
Keywords - distributed system, causality, vector time

1. INTRODUCTION

A distributed system is composed of multiple processes, executing on different processors, cooperating to solve a problem. The individual processes execute at the speed of the hosting processor and share no common memory. The sole means of communication is through asynchronous message passing with undetermined transmission delays.

While powerful in that large problems can more readily be solved, these distributed systems are both complex to implement and difficult to maintain. Causality among events of the system's execution provides a means of understanding the interaction of the constituent processes and the impact of a process's execution on other executing processes. Lamport's definition of the "*happens before*" relationship[19] is the basis for deriving the causal relationship among events. If an event e happens before event e', then event e can causally impact the execution of e'. Logical time and vector clocks[20, 12] provide a mechanism for partially ordering events according to their causal relationships.

Received November 1, 2006

Vector time and the happens before relationship are employed by a wide range of solutions for understanding distributed systems[7, 13, 2, 1]. Consider an event e of the system. Each other event in the system can be classified as concurrent to, causally succeeding or causal preceding event e. The events that causally succeed e can be causally impacted by the execution of e. Also for the events that causally precede e, their execution could have causally impacted the execution of e.

Although causality accurately models the possibility of one event affecting the execution of another, it does not provide for a quantitative measure of the impact. In other words, what is not known is the *timeliness* of events that causally precede and succeed e. Consider an event f that causally precedes e. If event f immediately precedes e, f not only happens before e but f is close in time to the occurrence of e. In contrast, consider the case where f's causal relationship to e is defined by a causal chain of hundreds of events. Although f may still have a causal impact on e, the importance of this causal relationship declines as the timeliness between the events increases.

As an example, consider a run-time error in a long running distributed execution. Suppose event f of process i, P_i, causally precedes the error. Additionally event f is the first executed event of P_i and the causal relationship to the error is defined by a long series of inter-process communications. Theoretically, from the definition of causality, event f could have caused the error. Knowing f preceded the error by several days may indicate that more timely events should be examined first. Concentration should be focused on events that are temporally closer to the error.

The timeliness of events that causally succeed e is also informative. Suppose the software of a process requires updating. After the update, testing follows. Specialized test cases may exist to understand the impact of the change, but it may be unknown how long each test case should be executed to reach a conclusion. Quantitative causality provides a means to label causally succeeding events so the potential impact of the change is distinguishable among these events.

This paper develops a quantitative measure of causality that indicates a temporal relevance between causally related events. The timeliness values can be used to identify those events that are more causally related than others. Computation of quantitative values follows Lamport's definition of causality. The calculation of the quantitative timeliness values is independent of both synchronized processor clocks and a global clock.

The necessary background to introduce quantitative causality is provided in section 2. Quantitative causality for succeeding events is defined in section 3 with weight calculation formulas. Section 4 provides a case study to demonstrate the use of quantitative causality. Conclusions are in section 5 while section 6 outlines future work.

2. BACKGROUND

A distributed system is composed of a set of N processes, $P_0 \dots P_{N-1}$. The execution of each P_i is seen as a totally ordered sequence of events, $e_i^0, e_i^1, \dots$, where event e_i^k is the k^{th} event executed in process P_i. The processes share neither a common memory nor synchronized clocks. Inter-process communication is through asynchronous message passing with unknown but finite transmission delays.

Assume that a message m is transmitted from process P_i to process P_j. Some event, e_i^k, on P_i is the transmission event of m, and some other event, e_j^l, on P_j will be the receipt event of m. Two functions given in definitions 1 and 2 will be used to refer to the communication partner event when only one endpoint is known. That is, $send(e_j^l) = e_i^k$ and $recv(e_i^k) = e_j^l$.

Definition 1 send(e)*: The event transmitting the message received by event e.*

Definition 2 recv(e)*: The event receiving the message transmitted by event e.*

2.1. Causality

Causality plays a major role in the operation of distributed systems. Some examples are rollback and recovery[28, 30, 29], global state reasoning[7, 13, 27, 17], debugging[10, 16], deadlock detection[4, 24, 15], termination detection[23, 9, 31], and software development and testing[25, 14, 18]. The "*happens before*" or *causal* relationship[19] is a means for defining order among events in a distributed system. This relationship is defined by a combination of the relative occurrence of events on a single process and by inter-process communication. If event e causally precedes event e', then the execution of e can have a causal impact on e'. Lamport formalized this relationship and its transitive closure as given in definition 3.

Definition 3 *Event e causally precedes event e', written $e \longrightarrow e'$, if and only if*

1. *events e and e' are executed in the same process and e temporally precedes e',*
2. *event e is the transmission of message m and e' is the receipt of the same message m, or*
3. *event $e \longrightarrow e''$ and $e'' \longrightarrow e'$.*

Causality is a partial order on the events of the system. Some events are not causally related but could occur in any order or simultaneously. The relationship given in definition 4 is referred to as *concurrency.* For any two events, e and e', if e does not happen before e' and e' does not happen before e, then e and e' are concurrent. Concurrency is a non-transitive relationship.

Definition 4 *Event e is concurrent to event e', written $e \| e'$, if and only if*

1. $e \not\rightarrow e'$ and

2. $e' \not\rightarrow e$

When a distributed system executes, a partial order among events is defined by the causal relationships. Consider the three process example of figure 1 where dashed lines indicate inter-process communication. Some of the causal relationships found in the represented execution are: $e_2^5 \rightarrow e_2^6$, $e_2^6 \rightarrow e_0^9$ and $e_2^5 \rightarrow e_0^9$. Note that $e_2^5 \rightarrow e_0^9$ exists from the transitivity property of causality. Concurrent relationships can also be found in this execution. For example, the absence of a causal relationship between events e_2^5 and e_1^7 implies that the events are concurrent: $e_2^5 \| e_1^7$. Other concurrent relationships are $e_0^1 \| e_2^1$ and $e_2^1 \| e_1^2$. However, observe that $e_0^1 \rightarrow e_1^2$ which demonstrates that concurrency is not transitive.

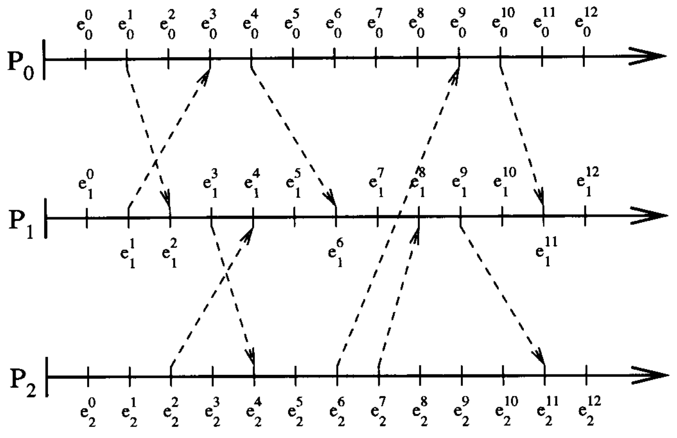

Figure 1: A three-process execution

For defining quantitative causality, an *anchor* event is used. The role of the anchor event is to fasten a point of reference. For example, if a software change is made and the succeeding events are to be quantified, the event corresponding to the execution of the software change becomes the anchor event.

2.2. Vector Time

One means of comparing events to determine the causal or concurrent relationship between them was defined independently by Mattern[20] and Fidge[12]. Integer vectors are used to represent the logical execution order of events. These vectors can also be used to accurately determine causality and concurrency among events.

Each P_i maintains a vector τ_i of N integers, $(\tau_i[0], \ldots, \tau_i[N-1])$, where $\tau_i[i]$ is the counter of the number of events which have occurred on P_i. The local component of the vector time, $\tau_i[i]$, is incremented before each event in P_i. Vector time element $\tau_i[i]$ is also referred to as the logical clock of P_i. The vector time assigned to event e is $\tau(e)$. Entry $\tau(e)[j]$ is the number of events in P_j that happens before e and that can causally affect the execution of e. The vector times are maintained across the system by piggybacking vector time values onto outgoing messages.

Initially, all components of vector times are zero as given in equation (1).

$$\underset{i=0}{\overset{N-1}{\forall}} \; \underset{j=0}{\overset{N-1}{\forall}} \; \tau_i[j] = 0 \tag{1}$$

When an event e_i^k occurs, the vector time on P_i is updated based on the event type of e_i^k, Equation (2) shows how vector times are modified to include piggybacked values from incoming messages. The updated vector time is attached to e_i^k and, if e_i^k is a send event, on the outgoing message. If e_i^k is a receive event, the vector clock on P_i is assigned the component-wise maximum of the local vector clock and the vector time attached to the incoming message.

$$\begin{array}{l} \tau_i[i] = \tau_i[i] + 1 \\ \text{if } \exists e : e_i^k = \mathit{recv}(e) \text{ then} \\ \quad \underset{j=0}{\overset{N-1}{\forall}} \; \tau_i[j] = \max(\tau_i[j], \tau(e)[j]) \end{array} \tag{2}$$

It has been shown that vector time is both sufficient and necessary to accurately model the causal relationships in a distributed execution[5]. A comparison of the vector times of two events will accurately describe the relationship between the events. Event e causally precedes event e' if the vector time assigned to e, $\tau(e)$, is less than the vector time assigned to e', $\tau(e')$. That is, $e \rightarrow e'$ if and only if $\tau(e) < \tau(e')$ using the vector comparison of equation (3).

$$\tau(e) < \tau(e') \iff \underset{j=0}{\overset{N-1}{\forall}} \tau(e)[j] \leq \tau(e')[j] \;\; \wedge \;\; \exists i : \tau(e)[i] < \tau(e')[i] \tag{3}$$

Vector time partitions events into three regions with respect to an anchor event. The regions, shown in figure 2, are events that precede the anchor, are concurrent to the anchor, and succeed the anchor. Preceding events fall into the shaded region on the left while succeeding events fall into the shaded region on the right. The events concurrent to the anchor event are in the non-shaded area.

Although causality is needed to reason about distributed systems, causality is an absolute relationship. Two events are either causally related or concurrent. Distributed executions are composed of many events and may be long running systems. The causality of events can be determined with mechanisms such as vector timestamps

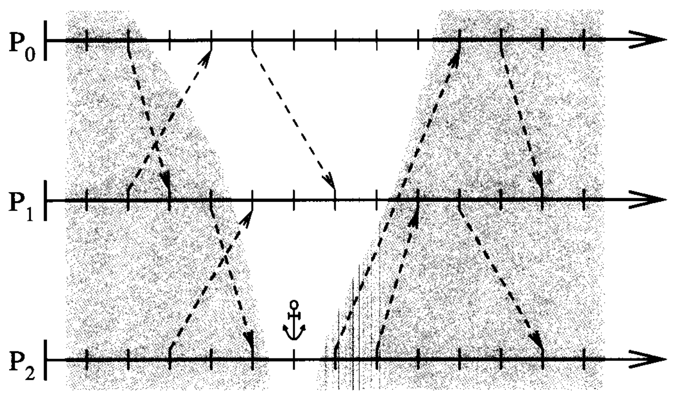

Figure 2: Regions of causality using vector time

but the timeliness of the causality is not captured. We define a model, Quantitative Causality for Succeeding Events (QCSE), for assigning weights to events where the weights signify the timeliness of the causality.

3. QUANTITATIVE CAUSALITY FOR SUCCEEDING EVENTS

Let event e_i^k be the anchor event. Events can be labeled as to whether they causally succeed e_i^k with the use of vector timestamps. All events whose vector time-stamp is greater than $\tau(e_i^k)$ causally succeed the anchor, i.e., event e causally succeeds e_i^k if $\tau(e_i^k) < \tau(e)$. This work introduces a weight, $\omega(e)$, which labels event e according to its causal relevance to the anchor event. The value assigned to event e quantifies the causal impact that the anchor event has on the execution of e. The range of weight values is from 1.0 to 0.0. A value of 1.0 is only assigned to the anchor event and a value of 0.0 suggests that the causal impact of e_i^k has diminished to a negligible amount.

In general terms, the weight of an event e_j^l is defined as a function on the event and the anchor as shown in equation (4).

$$\omega(e_j^l) = f(e_j^l, e_i^k) \tag{4}$$

The event weight $\omega()$ is dependent on function $f(\cdot)$ and allows flexibility in the means and rate of decrease from 1.0 to 0.0. Three definitions of $f(\cdot)$ follow and additional definitions can be developed according to the behavior of the system.

3.1. Piecewise Linear Decay

The first weight function models a piecewise-linear decrease in the assigned value as the temporal proximity to the anchor event, e_i^k, decreases. In other words, as the time between e_i^k and the other events increases, the weight values decrease. Computation is reliant on the local processor's clock value but requires neither inter-process synchronization of clocks nor common clock speeds.

The weight assigned to a causally succeeding event decreases as temporal distance to e_i^k increases. The rate of decrease is specified by the value T, the expected amount of time (in no particular unit) for the impact of e_i^k to diminish to a negligible amount. This value provides the means to customize the longevity of the impact of e_i^k. For a long running system, T may be relatively large compared to the value of T for a short running system.

Assume that some amount of time, δ (in the same units as T), elapses between the execution of any two consecutive events on a processor. A rate of decrement, α, is computed as shown in equation (5) to represent the minimum amount of weight decrease between any two consecutive events on a processor. The value of α ranges from 0.0 to 1.0 and is the amount of weight reduction for two consecutive events.

$$\alpha = \frac{\delta}{T} \tag{5}$$

For example, suppose we expect the impact of the anchor events to diminish in 2 seconds of execution time. Considering the local clock speed and the granularity of events, we determine that at least one microsecond will elapse between any two consecutive events. The piecewise-linear decay rate would be computed as follows.

$$\alpha = \frac{\delta}{T} = \frac{0.001 \text{ sec}}{2 \text{ sec}} = 0.0005$$

Events that are not found in the "*causally succeeding*" region of execution are assigned a value of 0.0. The anchor event is assigned a weight of 1.0. All other events, those causally following the anchor event, are assigned weights that are calculated from the immediately preceding event(s).

Local compute events and send events have a single immediate predecessor where the computation $f(\cdot)$ decreases the weight by α until a value of 0.0 is reached. However, when computing the weight of a receive event, there are two immediately preceding events to be considered. A receive event is assigned a weight that is either a decreased amount from the preceding event on the same process or the weight of the send event. The definition of $f(\cdot)$ as a piecewise-linear decay is given in equation (6).

$$f(e_j^l, e_i^k) = \begin{cases} 0.0 & \text{if } e_i^k \not\to e_j^l \\ 1.0 & \text{if } j = i \wedge l = k \\ \max(\omega(e_j^{l-1}) - \alpha), 0) & \text{if } e_j^l \text{ is not a receive event} \\ \max(\omega(e_j^{l-1}) - \alpha), \omega(\mathit{send}(e_j^l))) & \text{if } e_j^l \text{ is a receive event} \end{cases} \tag{6}$$

Consider the example execution shown in figure 3 where the computation of weights is based on an assumed value of $\alpha = 0.1$ and the indicated anchor event. Each event is labeled with the assigned weight. Events that do not causally follow the anchor event are assigned a weight of zero. These weights are faded to focus attention on the non-zero weight propagation.

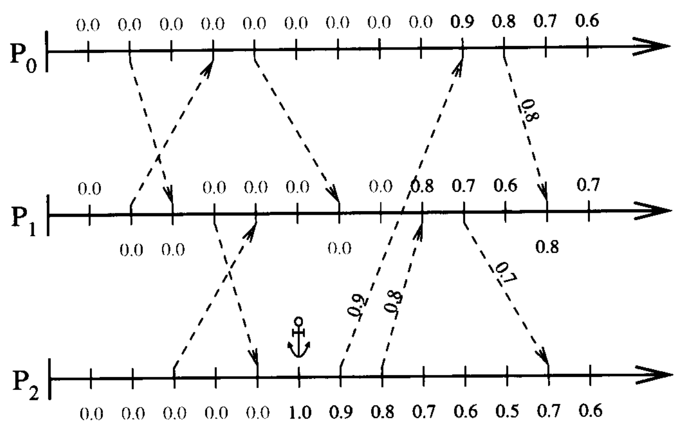

Figure 3: Piecewise linear weight decrease with $\alpha = 0.1$

The weight assigned to the anchor event is 1.0. Immediately following the anchor in process P_2 is an event whose weight is 0.9, computed as $1.0 - 0.1$. Both the event and the message that is sent from the event are labeled with the value 0.9. When the message arrives at P_0, the attached weight is used to compute the value of the receive event. The receive event is assigned the maximum of the decremented weight of the immediately preceding events on P_0 $(0.0 - 0.1 = -0.1)$ or the message weight (0.9).

Each of the first five events on P_2 that causally follow the anchor have weights that decrease by α from the immediate predecessor event. However, the value assigned to the sixth event, a receive event, increases compared to its immediate predecessor. The reason for this is that the assigned value is taken to be the maximum of the value of the decremented weight of the immediately preceding events on P_2 $(0.5-0.1 = 0.4)$ or the message weight (0.7).

As the system executes, the weights assigned to events decrease. However, the decrease is not monotonic on a process since a receive can cause a local increase in event weight. Our previous work[26] proved that the weight assigned to events on each process reaches zero in a finite amount of time. To bound the time, assumptions of the upper bounds on inter-event delays and communication latency were made.

3.2. Exponential Decay

Depending on the situation and use of the event weights, a linear decay may not be the most applicable decrease function. It is foreseen that cases will arise where weight of events should decrease but never be allowed to diminish to zero. In these cases, an exponential decay may be more appropriate.

As in the linear decay, a value is defined that controls the rate at which the decay occurs. The value of β ranges from 1.0 to 0.0. Values closer to 1.0 give a quicker decreasing weight function, while values closer to 0.0 provide for effects that linger for a longer time.

$$f(e_j^l, e_i^k) = \begin{cases} 0.0 & \text{if } e_i^k \not\rightarrow e_j^l \\ 1.0 & \text{if } j = i \wedge l = k \\ \frac{\omega(e_j^{l-1})}{1+\beta} & \text{if } e_j^l \text{ is not a receive event} \\ \max(\frac{\omega(e_j^{l-1})}{1+\beta}, \omega(send(e_j^l))) & \text{if } e_j^l \text{ is a receive event} \end{cases} \tag{7}$$

Exponentially decreasing weights are assigned to events in a manner similar to the linear decrease from the previous section. Equation 7 shows that weights assigned to events not succeeding the anchor are zero while the anchor is assigned a weight of one. Events causally following the anchor are assigned weights of $\frac{1}{(1+\beta)^1}$, $\frac{1}{(1+\beta)^2}$, $\frac{1}{(1+\beta)^3}$, and so on.

The weight decreases but, ignoring precision limits, will never reach zero. As in the linear decay computation, receipt of a message from an event with a higher weight will increase the weight on the local process. Figure 4 shows the weight decrease using several values of β, ranging from 0.01 to 0.10 in increments of 0.01. The graph does not show the weight increase caused by incoming messages but assumes that no external messages arrive with a higher weight than the one locally computed. The horizontal scale represents the number of events that are present between the current event and the anchor event.

This distribution of weights will correctly model systems where the effects of events are immediate in most cases, but can linger for an indeterminate amount of time. It could be the case that the last event in a long running system could be causally impacted by the first event in the system.

Consider the example execution shown in figure 5 where the computation of weights is based on an assumed value of $\alpha = 0.1$ and the indicated anchor event. Each event is labeled with the assigned weight. Events that do not causally follow the anchor event are assigned a weight of zero. These weights are faded to focus attention on the non-zero weight propagation.

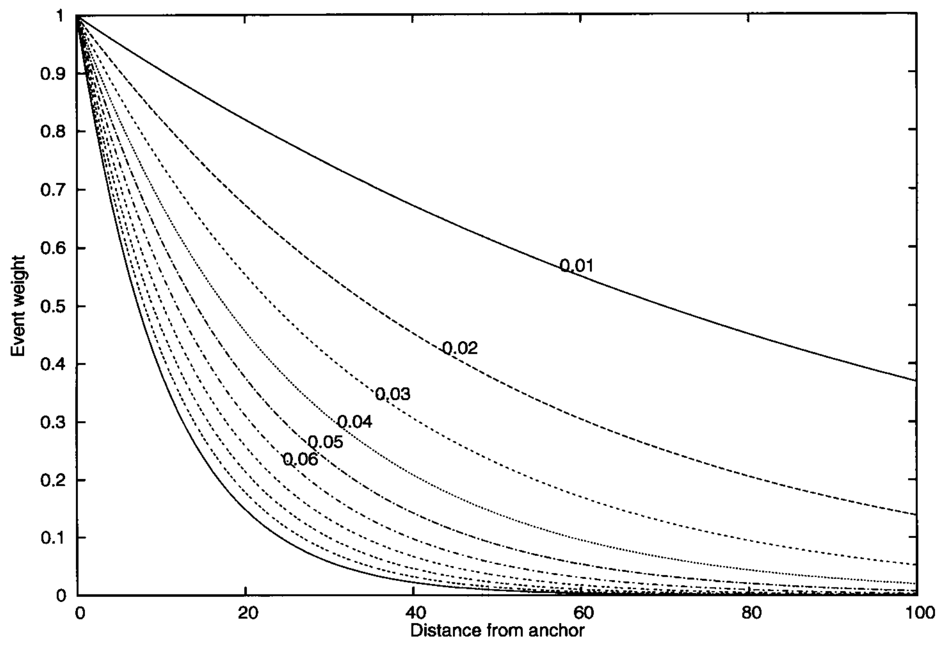

Figure 4: Exponential decay with values of β between 0.01 and 0.10

3.3. Vector Difference

The next definition of the weight function is reliant on a modified version of vector timestamps, a *relative vector* π of N integers. Only the anchor event and events that causally succeed the anchor will be labeled with a relative vector. The relative vector is for determining how many events have occurred on each process since e_i^k. For event e_j^l, where $e_i^k \to e_j^l$, the relative vector of e_j^l, $\pi(e_j^l)$, captures the number of events that have occurred on each process after the execution of e_i^k and up to the execution of e_j^l.

The π for the anchor event e_i^k is set to all zeroes as given in equation (8).

$$\overset{N-1}{\underset{j=0}{\forall}} \pi_i[j] = 0 \tag{8}$$

Other processes $P_j, j \neq i$, do not have a π until one is propagated to it through inter-process communication. Processes that possess a relative vector update the values as shown in equation (9). Message transmission events will effectively piggyback the relative vector on all outgoing messages.

$$\begin{aligned} &\text{if } \exists e : e_j^l = recv(e) \text{ then} \\ &\quad \overset{N-1}{\underset{m=0}{\forall}} \pi_j[m] = \max(\pi_j[m], \pi(e)[m]) \\ &\pi_j[j] = \pi_j[j] + 1 \end{aligned} \tag{9}$$

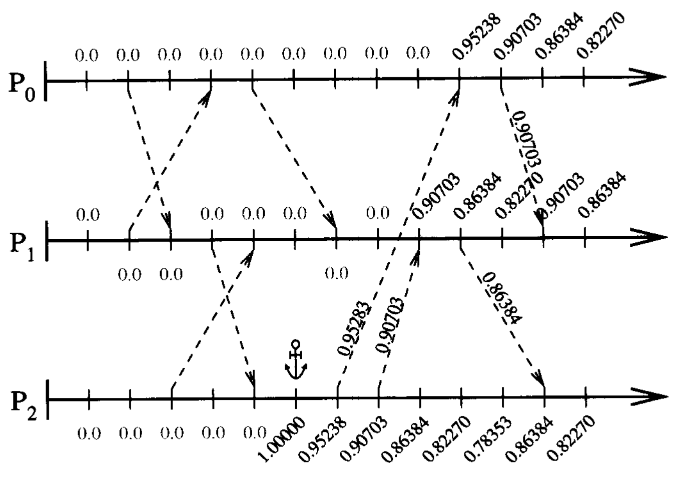

Figure 5: Exponential weight decrease with $\beta = 0.05$

The relative vector captures the total number of events that the anchor event has impacted. A parameter Π is a bound on the impact of the anchor event and states that after Π events have occurred, the effects of the anchor event will be negligible. The value of Π will vary according to the distributed system and the nature of the anchor event. From the $\pi(e_j^l)$ and the parameter Π, the weight for event e_j^l is computed.

$$f(e_j^l, e_i^k) = \begin{cases} 0.0 & \text{if } e_i^k \not\to e_j^l \\ 1.0 & \text{if } j = i \wedge l = k \\ \max(\frac{\Pi - \sum_{k=0}^{N-1} \pi(e_j^l)[k]}{\Pi}, 0) & \text{otherwise} \end{cases} \tag{10}$$

Figure 6 provides an example to demonstrate the use of the above equation. The figure has labels on events that contain both the relative vector attached to the event and the computed weight of the event. Events that do not have a relative vector are labeled with [-,-,-] that is faded.

The anchor event on P_2 is given a relative vector of all zeroes which results in a computed weight of 1.0. The relative vector is incremented in the local component for the next event on P_2. A resulting weight is computed as $\frac{10-1}{10} = 0.9$ indicating a high relative significance to the anchor. Since the event is a message transmission, the vector assigned to the event is also attached to the outgoing message. When the message is received at P_0, it is used to update the relative vector of the receive event which is, in turn, used to compute the weight of the receive event.

Although this technique, as presented, incurs excessive overhead for storage and computation of a secondary vector timestamp, this technique can be implemented as a computed vector derived from traditional vector timestamps. For example, the

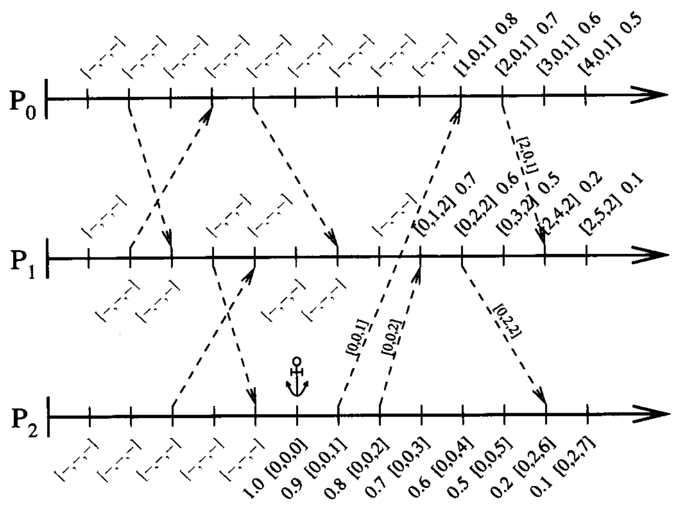

Figure 6: Relative vector timestamps and event weights with $\Pi = 10$

relative vector of an event can be computed from the vector timestamps of the anchor event and the event in question. Equation (11) shows the computation of the relative vector given that $e_i^k \rightarrow e_j^l$. No additional run-time expense is incurred as the computation can be made post mortem.

$$\overset{N-1}{\underset{m=0}{\forall}} \pi(e_j^l)[m] = \tau(e_j^l)[m] - \tau(e_i^k)[m] \tag{11}$$

As with the piecewise linear decay, excluding the effects of arriving messages, vector difference will provide a linear decrease in event weights. In contrast to the piecewise linear decay, weights are monotonically decreasing on each process. A receive event will not result in a weight increase from the immediate preceding events. The arrival of a message may increase the components of the relative vector, but the vector increase results in a lower computed weight.

4. CASE STUDY

To better understand the importance of quantitative causality, we examined a problem common to software engineers - the *feature location problem*[3, 32, 6, 8]. For a distributed program, a feature location study has been conducted with and without quantitative causality.

Features are services that the software provides to its users. Consider a word processor. Examples of features are *word wrap*, *spell check*, *font change*, etc. To the user, the features appear to be a single action. However, the implementation may involve many different software components cooperatively resolving the request. An error report or upgrade request initiated by the user is commonly in terms of features. Software engineers need to understand the code involved in the feature in order to make the necessary changes. Locating the source code that implements a particular user accessed feature of the software is non-trivial. An understanding of the code is also needed before any additional features are added to reduce the chance of disturbing existing functionality.

We conducted a case study[11] where the focus was providing a methodology for the feature location problem. Software components are ranked according to a calculated value to indicate the component's relevance to the feature under investigation. In this study, the usefulness of quantitative causality is illustrated by comparing the exclusion and inclusion of quantitative causality in the ranking of components. Only the portions of the study relevant to quantitative causality are presented.

4.1. Gunner

The case study used a distributed program entitled *Gunner*. The Gunner program is a simple text-based game developed using MPI[21, 22] as a programming exercise in several courses. It simulates a medieval gunner firing a cannon at a castle. The program has two main features: *move the gun* and *take a shot*.

The software components that implement the *move the gun* feature, F, were sought. When the system was executed, 169 software components were traced. The methodology ranks the components from highest to lowest based on traced executions of a feature. The ranking distinguishes the more relevant components of the feature so that key components can be identified.

Fundamental to locating the key software components is the association between events of a distributed system's execution and the components of the distributed system. A software component, c_q, is the source of an event, e_i^k, if event e_i^k was the direct result of the execution of c_q. The function $\sigma()$ maps the event to the source component.

$$c_q = \sigma(e_i^k) \tag{12}$$

Figure 7 shows the system model. On the left is the collection of software components that gives a static view of the system. On the right is the dynamic view of the system as it is defined by a particular execution. A function σ maps a dynamic event to the static component. Although each event will map to a single component, the opposite is not the case. Consider the multiple events generated when a component is executed inside the body of a loop.

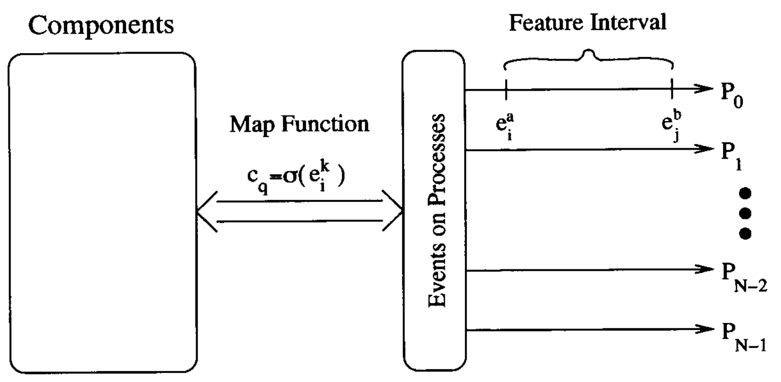

Figure 7: The system model

A feature is typically not one event but several events, termed an *interval of events* for feature F. Relating this back to the concept of an anchor event, instead of one events constituting an anchor, the multiple feature events collectively form an anchor. An interval consists of a start event, e^a, an end event e^b, and all events that both causally follow the start event and causally precede the end event. Multiple intervals are taken from multiple executions of the feature. For example, t executions of the feature would form the set I^*.

$$I^* = I_1 \cup I_2 \cup ... \cup I_t \tag{13}$$

From the intervals, a *component relevance index*, $\hat{p}_c$ is calculated. The value of $\hat{p}_c$ will range from 0.0 to 1.0 and is the numeric value for ranking the component.

Initially in the case study, each event in the interval was assigned a weight of one and all other events are assigned a weight of zero.

$$\omega(e) = \begin{cases} 1 & \text{if } e \in I^* \\ 0 & \text{if } e \notin I^* \end{cases} \tag{14}$$

The value of $\hat{p}_c$ is calculated as:

$$\hat{p}_c = \frac{\sum\limits_{e:c=\delta(e)} \omega(e)}{|\{e : c = \delta(e)\}|} \tag{15}$$

For the case study of Gunner and the feature *move the gun*, the feature was repeatedly executed until $\hat{p}_c$ rankings stabilized. The top 20 of 169 components are shown in table 1. The source code was instrumented at function entry and exit as well as MPI function calls.

There is a clear division of components identified by the difference in $\hat{p}_c$ values of 0.8885 and 0.1600. Nine of the key components have $\hat{p}_c$ values that indicate they are

Function	**Event Type**	$\hat{p}_c$
gunner_display	FEntry	1.0000 ± 0.0000
gunner_display	FExit	1.0000 ± 0.0000
moveGun	FEntry	1.0000 ± 0.0000
moveGun	FExit	1.0000 ± 0.0000
moveGun	Send	1.0000 ± 0.0000
validate_gunner	FEntry	1.0000 ± 0.0000
validate_gunner	FExit	1.0000 ± 0.0000
doEmptyLayoutWithGunner	FEntry	0.8885 ± 0.0368
doEmptyLayoutWithGunner	FExit	0.8885 ± 0.0368
Process4	Recv	0.1600 ± 0.1260
setGunner	FEntry	0.1204 ± 0.1024
writeCharRows	FEntry	0.1006 ± 0.0344
display	FExit	0.1003 ± 0.0344
Graphics	FEntry	0.1003 ± 0.0344
Graphics	FExit	0.1003 ± 0.0344
showBorders	FEntry	0.1003 ± 0.0344
showBorders	FExit	0.1003 ± 0.0344
writeCharRows	FExit	0.1003 ± 0.0344
writeHorizontalBorder	FEntry	0.1003 ± 0.0344
writeHorizontalBorder	FExit	0.1003 ± 0.0344

Table 1: The top 20 components of `Gunner` ranked by $\hat{p}_c$

part of the *move the gun* feature. A problem was found: two other components exist that are key to the feature but are not highly ranked. The reason for the omission of key components is that events that causally follow the start event but are not part of the interval are excluded, i.e., assigned a weight of zero. The events that causally follow e^a and happen in some reasonable short time after the interval should be considered.

The need for quantifying events was realized so a more accurate ranking could be accomplished. The addition of a piecewise linear decay was employed. Three experiments were conducted with T set to 0.5 seconds, 2.0 seconds, and 5.0 seconds. The correct 11 components are found in all three experiments. The $\hat{p}_c$ values shown in table 2 were calculated.

Function	Event Type	0.0 sec	0.5 sec	2.0 sec	5.0 sec
gunner_display	FEntry	1.0000	1.0000	1.0000	1.0000
gunner_display	FExit	1.0000	1.0000	1.0000	1.0000
moveGun	FExit	1.0000	1.0000	1.0000	1.0000
moveGun	FEntry	1.0000	1.0000	1.0000	1.0000
moveGun	Send	1.0000	1.0000	1.0000	1.0000
validate_gunner	FExit	1.0000	1.0000	1.0000	1.0000
validate_gunner	FEntry	1.0000	1.0000	1.0000	1.0000
doEmptyLayoutWithGunner	FEntry	0.8885	0.8885	0.8885	0.8889
doEmptyLayoutWithGunner	FExit	0.8885	0.8885	0.8885	0.8889
Process4	Recv	0.1600	0.9950	0.9950	0.9950
setGunner	FEntry	0.1204	0.7405	0.8635	0.8882
writeCharRows	FEntry	0.1006	0.1006	0.1026	0.1352
display	FExit	0.1003	0.1003	0.1022	0.1335
Graphics	FExit	0.1003	0.1003	0.1028	0.1372
Graphics	FEntry	0.1003	0.1003	0.1028	0.1372
showBorders	FExit	0.1003	0.1003	0.1028	0.1372
showBorders	FEntry	0.1003	0.1003	0.1028	0.1372
writeCharRows	FExit	0.1003	0.1003	0.1023	0.1342
writeHorizontalBorder	FExit	0.1003	0.1003	0.1023	0.1342
writeHorizontalBorder	FEntry	0.1003	0.1003	0.1024	0.1349

Table 2: Identifying components in Gunner using $\hat{p}_c$ values for different decay times

5. CONCLUSIONS

The well-defined tertiary relationships between a pair of events in the executions of a distributed system are either "*preceding*", "*succeeding*", or "*concurrent*". We have developed a novel approach for event comparison that provides a quantitative measure of the timeliness of events in conjunction with the causal relationships. *Quantitative causality* assigns weights to events to quantify the timeliness of the events. This paper focuses on events that causally succeed an anchor event. The QCSE model provides a mechanism for assigning weights to events.

In QCSE, events that occur within close temporal proximity are given a greater weight than events that occur further apart in time. Computation of event weights relies on a function that is customizable according to the type of system being executed and the purpose of the weights. Three weight functions are provided. Each function is parametrized to control the rate of decrease. This parametrization allows tailoring the weights to a particular distributed system behavior. For example, the parametrization can be customized for the running time of the system or the longevity of an event's impact.

Linear decreasing weights are provided by the piecewise linear decay function. A parameter determines the slope of the decay. The function is piecewise in that the arrival of a message could convey more timely information and cause the local weight to increase before returning to the linear decrease.

A parametrized Exponential decay was defined to provide a lingering weight that approaches zero. We expect this weight function to more accurately represent the declining relevance of causal relationships in some system executions. The rate of decrease is controlled by a single parameter with the effects of example values shown. Arriving message can cause the local event weight to increase.

A vector time that is relative to the anchor event is used to compute weights for events in the third function. The vector tracks the number of events in all processes that happen between the anchor event and the current event. A proportion function is applied to the relevant vector to determine the weight to be assigned. Arriving messages will cause the weight to decrease providing a monotonically decreasing function. The parameter determines the slope of the decay functions.

To demonstrate the utility of quantitative causality, we provide a case study centered around the feature location problem. The study involves a small distributed game called *Gunner.* The game was analyzed to locate a specific feature. The analysis was performed in two different ways: once without using quantitative causality and then with quantitative causality. Quantitative causality enabled the identification of all key components where the original feature location analysis failed.

5.1. Future Work

The QCSE model propagated weights for events succeeding an anchor event. Another model, Quantitative Causality for Preceding Events (QCPE), is currently being developed to assign weights to events that causally precede the anchor event. This will be analogous to QCSE in that weights will reflect the timeliness of the events to the anchor. If the anchor event is the symptom of a *bug*, the quantifying of preceding events could form a set of those first implicated as the source of the error and rank the possible cause locations. Similar functions, piece-wise linear, exponential and vector difference, should be appropriate for this model.

To complete the triad of distributed system event relationships we will explore the use of quantitative measures applied to concurrent relationships. Quantifying the concurrency of an event with respect to an anchor event would provide a means to reason globally about the state of the system. An immediate challenge with quantitative concurrency is that concurrency is not transitive. Assigned weights can not be propagated as they are in the causal model. Regions of concurrency with respect to an anchor event can be identified and the weights computed over that region.

A simple example of quantitative concurrency considers two concurrent events from different processors that, if executed simultaneously, could cause deadlock. The definition of concurrency states whether or not the possibility exists for simultaneous execution, not how likely the occurrence. We will develop a means of assigning a weight to pairs of concurrent events to indicate the likelihood of simultaneity so the most likely source of errors can be the focus.

Potential case studies to examine the implementation of assigned weights are under consideration. One, receiving current funding from Northrup-Grumman, will examine distributed system traces using quantitative causality and concurrency metrics for design recovery.

REFERENCES

[1] Emmanuelle Anceaume, Jean-Michel Hélary, and Michel Raynal. Tracking immediate predecessors in distributed computations. In *Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*, pages 210–219. ACM Press, 2002.

[2] Anish Arora, Sandeep Kulkarni, and Murat Demirbas. Resettable vector clocks. In *Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing*, pages 269–278. ACM Press, 2000.

[3] Ted Biggerstaff, Bharat Mitbander, and Dallas Webster. Program understanding and the concept assignment problem. *Communications of the ACM*, 37(5):72–83, May 1994.

[4] K.M. Chandy, J. Misra, and L. Haas. Distributed deadlock detection. *ACM Transactions on Computer Systems*, 1(2):144–156, 1983.

[5] Bernadette Charron-Bost. Concerning the size of logical clocks in distributed systems. *Information Processing Letters*, 39:11–16, July 1991.

[6] Kumrong Chen and Vaclav Rajlich. Case study of feature location using dependence graph. In *Proceedings of the 8th International Workshop on Program Comprehension - IWPC 2000*, pages 241–249, Los Alamitos, California, June 2000. IEEE Computer Society.

[7] R. Cooper and K. Marzulo. Consistent detection of global predicates. In *Proceedings of the ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*, pages 163–173, 1991.

[8] J. Deprez and A. Lakhotia. A formalism to automate mapping from program features to code. In *Proceedings of the 8th International Workshop on Program Comprehension - IWPC 2000*, pages 69–78, Los Alamitos, California, June 2000. IEEE Computer Society.

[9] E. Dijkstra, W. Feijen, and A. van Gasteren. Derivation of a termination detection algorithm for distributed computations. *Information Processing Letters*, 16:217–219, 1983.

[10] P. S. Dodd and C. V. Ravishankar. Monitoring and debugging distributed real-time programs. *Software Practice and Experience*, 22(10):863–877, October 1992.

[11] Dennis Edwards, Sharon Simmons, and Norman Wilde. An approach to feature location in distributed systems. *Journal of Systems and Software*, 79(1):57–68, January 2006.

[12] C. J. Fidge. Partial orders for parallel debugging. In *Proceedings of the ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*, pages 183–194, University of Queensland, May 1988.

[13] K. K. Garg and C. Chase. Distributed algorithms for detecting conjunctive predicates. In *International Conference of Distributed Computing Systems*, pages 423–430, June 1995.

[14] A. Gravey and A. Dupuis. Performance evaluation of two mutual exclusion distributed protocols via markovian modeling. In *Proceedings of the Sixth IFIP Workshop on Protocol Specification, Testing, and Verification*, pages 335–346, 1987.

[15] Mehdi Hashemzadeh, Nacer Farajzadeh, and Abolfazl T. Haghighat. Optimal detection and resolution of distributed deadlocks in the generalized model. In 14th *Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 133–136, Feb 2006.

[16] Curtis E. Hrischuk and C. Murray Woodside. Logical clock requirements for reverse engineering scenarions from a distributed system. *IEEE Transactions on Software Engineering*, 28(4):321–339, Apr 2002.

[17] J. Janb. Performance features of global states based application control. In 14th *Euramicro International Conference on Parallel, Distributed, and Network-based Prcessing*, pages 276–279, Feb 2006.

[18] Ahmed Khoumsi. A temporal approach for testing distributed systems. *IEEE Transactions on Software Engineering*, 28(11):1085–1103, Nov 2002.

[19] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[20] Friedemann Mattern. Virtual time and global states of distributed systems. In M. Cosnard et. al., editor, *Proceedings of the International Workshop on Parallel and Distributed Algorithms*, pages 215–226. Elsevier Science Publishers B. V., 1989.

[21] Message Passing Interface Forum. MPI: A message-passing interface standard (version 1.1). Technical report, MPI-Forum, http://www.mpi-forum.org, 1995.

[22] Message Passing Interface Forum. MPI-2: Extensions to the message-passing interface. Technical report, MPI-Forum, http://www.mpi-forum.org, July 1997.

[23] S. Rana. A distributed solution of the distributed termination problem. *Information Processing Letters*, 17:43–46, 1983.

[24] B.A. Sanders and P.A. Heuberger. Distributed deadlock detection and resolution with probes. In J-C. Bermond and M. Raynal, editors, *Proceedings of the Third International Workshop on Distributed Algorithms*, number 392 in Lecture Notes in Computer Science, pages 207–218. Springer-Verlag, 1989.

[25] Werner Schutz. Fundamental issues in testing distributed real-time systems. *Real-Time Systems*, 7(2):129–157, September 1994.

[26] Sharon Simmons and Dennis Edwards. Convergence of time decay for event weights. In *The 2006 International Conference on Parallel & Distributed Processing Techniques & Applications*, Las Vegas, Nevada, June 2006. World Academy of Science. To appear.

[27] Sharon Simmons and Phil Kearns. A causal assert statement for distributed systems. In M. H. Hamza, editor, *Proceedings of the Seventh IASTED/ISMM International Conference on Parallel and Distributed Computing and Systems*, pages 495–498. IASTED/ISMM, IASTED-ACTA Press, October 1995.

[28] R. Strom and S. Yemini. Optimistic recovery in distributed systems. *ACM Transactions on Computer Systems*, 3(3):204–226, 1985.

[29] Tongchit Tantikul and D. Manivannan. A communication-induced checkpointing and asynchronous recovery protocol for mobile computing systems. In *Sixth International Conference on Parallel and Distributed Computing, Applications and TRechnologies*, pages 70–74, Dec 2005.

[30] S. Venkatesan and Tony T-Y. Juang. Efficient algorithms for optimistic crash recovery. *Distributed Computing*, 8:105–114, 1994.

[31] Xinli Wang and Jean Mayo. A general model for detecting distributed termination in dynamic systems. In *Proceedings of the* 18^{th} *International Parallel and Distributed Processing Symposium*, page 84, Apr 2004.

[32] Norman Wilde and Michael Scully. Software reconnaissance: Mapping program features to code. *Journal of Software Maintenance: Research and Practice*, 7:49–62, January 1995.