

Convergence of Time Decay for Event Weights

Sharon Simmons and Dennis Edwards

Department of Computer Science, University of West Florida
11000 University Parkway, Pensacola, FL, USA

Abstract

Events of a distributed system are partially ordered, but additional ordering information can aid in understanding the causal impact of events on other events. The assignment of weights to events measures the temporal proximity of events and quantitates the causality among events. In particular, weights have been used in the feature location problem to determine the relevance of events to the feature. Weights propagate to other processes following the Lamport causal ordering, with an applied decay representing their time-distance from the point of certain relevance.

The focus of this paper is proving the decreasing property of weights as the time-distance from preceding events increases. Specifically, we prove that a weight of zero is reached despite intermediate weight increases that may incur from interprocess communication. An upper bound is placed on the elapsed time required before an event is assigned a weight of zero.

Keywords: distributed system, execution trace, event weight, causality, feature location

1 Introduction

For distributed systems, the concept of a partial order[2] is used to understand the ordering of events. The partial order can be established by examining the causal relationships stemmed from message passing between processes. However, in some circumstances, it is useful to provide a weighting of the *closeness* of events in time. Such a measure is useful in addressing the feature location problem in distributed systems[4].

Feature location is a well known problem faced by Software Engineers in understanding large software systems[7, 8, 6, 9, 1, 5]. Such systems may involve hundreds of thousands of lines of code and provide many different features to their users. The code for a particular feature is often not well localized, but

is spread around several different modules. Since comprehensive understanding of the entire system is not feasible, the Software Engineer instead locates and studies the code for the particular feature that needs to be fixed or enhanced.

Most feature location methodologies involve running the program and comparing a time interval in which the user invoked the feature with an interval in which the feature was inactive. Traces of execution from the two intervals are compared to look for code components that were only executed when the feature was present. Application of such methodologies to distributed systems is complicated by the stochastic behavior of such systems and by difficulties in determining the start and end points of feature intervals across processes.

A recently proposed feature location methodology for distributed systems assigns weights to events to indicate their relevance to the feature. Events that are known to be in the feature interval are given a weight of one. Weights propagate to other processes following the Lamport causal ordering, with a decay applied to represent their temporal distance from the point of certain relevance[4].

Propagating weights provides a quantitative measure of cause and effect in the causal ordering. The properties of such weights are thus of general interest, as well as of practical importance in feature location. An important property is that the weights should eventually decrease to zero for any possible pattern of inter-processor messaging.

The assignment of weights to events for the feature location problem and then more generally, the establishment of event weights decreasing to zero are the topics of this paper. In section 2, our system model and the assignment and propagation of weights is defined. In section 3, the proof of convergence of the weights is provided.

2 System Model and Component Relevance Index

A distributed system is modeled as a set, $C = \{c_1, c_2, \dots, c_q, \dots, c_Q\}$, of *software components*. Fundamental to this research is the association between events of a distributed system's execution and the components of the distributed system. An event is a discrete occurrence on a process. As software components execute, they generate events in the N different processes of the system, $P_1 \dots P_N$. Event e_i^k is the k_{th} event generated in P_i . The events on process P_i are identified by the set E_i . The set E contains all events in the system and is the union of the events from each individual process, i.e., $E = E_1 \cup E_2 \cup \dots \cup E_N$.

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 $\sigma()$ is a surjective map from an event to a source software component. That is, $c_q = \sigma(e_i^k)$ if c_q is the source of e_i^k . One source component can map to many events but an event maps to exactly one component.

Figure 1 depicts our system model. The left side of the figure shows the code components. These code components are executed giving rise to events in different processes, as shown in the time lines on the right. The mapping between the two is provided by the function $\sigma()$.

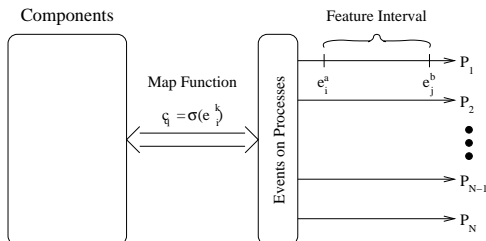


Figure 1: The System Model

The causal, or happens before, relationship[2] is one means of defining order among events in a distributed system. This relationship is defined either by the temporal occurrence of events on a single process or by interprocess 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 follows. Event e causally precedes event e' , written $e \rightarrow e'$, iff

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 $e \rightarrow e''$ and $e'' \rightarrow e'$.

We let F denote the feature being investigated. The software engineer gains information for mapping the feature F to software components by executing the feature. Each execution of a feature is associated with an *interval* of events defined by the start event of the feature, e_i^a , and the end event of the feature, e_j^b . The interval consists of the start event, the end event, and all events that both causally follow the start event and causally precede the end event. The events that occur some *reasonably* short time after the interval are also important for the feature location problem. Since, in the general case, effects of the feature outlast the duration of the feature.

The events of the interval are precisely defined, but the events that occur some reasonably short time after the interval need further identification. A weight function $\omega()$ has been developed to label events according to their relevance to the interval[4]. The weight assigned to an event decreases as temporal proximity to the interval increases. The value *decayTime* is the expected amount of time for the impact of the feature to diminish to a negligible amount. From this value, a *decay* factor is computed to indicate the amount a weight is reduced per unit of time.

$$decay = \frac{1}{decayTime} \quad (1)$$

Weights originate in the interval by assigning internal (to the interval) events a weight of one as shown in equation (2a). These weights are decreased and propagated to external (to the interval) events according to the definition of causality. Let $\tau(c)$ be the local clock time of event e . Equation (2b) shows that weights are computed from the weights of causally preceding events from the same process for events other than message receipts. The computation of weights for receive events is given in equation (2c). Note that a message *send* can propagate its weight to other processes without decay. However, all other event weights are diminished during propagation.

The weights of the events are used to compute a component relevance index[3, 4] for ranking components for a particular feature. The component relevance index will be close to one when the events of the component occur within the interval or close to the interval. The details of computing the component relevance index and case studies performed

can be found in [4]. This paper will show that the weights decay to zero so that the component relevance index will approach zero for components whose events occur a reasonably long time after the feature.

$$\omega(e_i^k) = \begin{cases} 1.0 & \text{if } e_i^k \in I \\ \max(\omega(e_j^{k-1}) - ((\tau(e_j^k) - \tau(e_j^{k-1})) \times \text{decay}), 0) & \text{if } e_i^k \text{ is not a receive and } e_i^k \notin I \\ \max(\omega(e_j^{k-1}) - ((\tau(e_j^k) - \tau(e_j^{k-1})) \times \text{decay}), \omega(\text{send}(e_j^k))) & \text{if } e_i^k \text{ is a receive and } e_i^k \notin I \end{cases} \quad (2a)$$

$$(2b)$$

$$(2c)$$

3 Proof

Although weights assigned to events on a process will be reduced to zero as time from the interval increases, the weights are not monotonically decreasing. A message receipt can cause the process weight to increase. However, every process's weight will reach zero in a finite amount of time. The amount of time is stated for each process P_i and is measured from the last event of the interval that causally impacts process P_i . We prove that the amount of time for each process's weight to reach zero is finite and place a maximum bound on this time.

A labeled, directed graph G provides the foundation for showing the weight reduction to zero. Graph G represents the events of the execution and conveys the causal relationships between those events.

Each event in the execution is represented by a unique node in G . We label each node to indicate the event from which it is derived. Node n_i^k represents the k^{th} event of process P_i , e_i^k .

$$n_i^k \in G \Leftrightarrow e_i^k \in E$$

We make the assumption that receive events and send events are distinct. That is, no single event can simultaneously receive a message and send a message.

Assumption 1 *Receive events and send events are distinct.*

The directed arcs of graph G represent the immediate causality between events. Transitively defined causal relationships are omitted. We categorize the arcs as either *intraprocess* or *interprocess*.

Definition 1 (Intraprocess) *If e_i^k and e_i^{k+1} are consecutive events on P_i such that $e_i^k \rightarrow e_i^{k+1}$, then arc $(n_i^k, n_i^{k+1}) \in G$*

Definition 2 (Interprocess) *If e_i^k is a message receipt event and $e_j^l = \text{send}(e_i^k)$, then arc $(n_j^l, n_i^k) \in G$*

Assumption 1 combined with definitions 1 and 2 describe the degrees of each node in the graph. A node representing a send event has an in-degree of 1 and an out-degree of two. Conversely, a node representing a receive event has an in-degree of two and an out-degree of one. Nodes representing events that are neither send nor receive events have both an in-degree and an out-degree of one.

The label of the arc connecting node n_i^k to n_j^l , denoted $v_{i,j}^{k,l}$, indicates the potential amount the weight of e_j^l is reduced from the weight of e_i^k . We first consider intraprocess arcs. Referring to figure 3, if the weight of event e_1^0 is given as $\omega(e_1^0)$, then the non-negative weight of event e_1^1 will be $\omega(e_1^0) - v_{1,1}^{0,1}$. More generally, for events represented by nodes with an in-degree of one, $\omega(e_i^{k+1}) = \max(\omega(e_i^k) - v_{i,i}^{k,k+1}, 0)$.

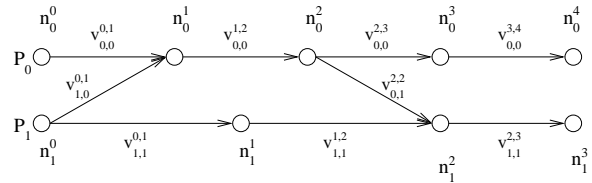


Figure 2: Two processes

We assume clocks are sufficiently precise to capture the elapsed time between two causally related events on the same process. We further assume that a minimum amount of time, Δ , elapses between any two causally related events.

Assumption 2 *At least Δ time will elapse between any two consecutive events on P_i .*

$$\tau(e_i^{k+1}) - \tau(e_i^k) \geq \Delta$$

Equation (2b), gives us that an intraprocess arc is assigned $v_{i,i}^{k,k+1} = (\tau(e_i^{k+1}) - \tau(e_i^k)) \times \text{decay}$. Using the inequality of assumption 2, we can conclude that $v_{i,i}^{k,k+1} \geq \Delta \times \text{decay}$.

Property 1 *The minimum value of any arc representing intraprocess causality is $\Delta \times \text{decay}$.*

$$v_{i,i}^{k,k+1} \geq \Delta \times \text{decay}$$

We next consider interprocess arcs. Two arcs point to a receive event: an intraprocess arc and an interprocess arc. In figure 2, arcs from both n_0^0 (intraprocess) and n_1^0 (interprocess) point to n_0^1 , a receive event. An interprocess arc is labeled with zero since the weight of the receive event is never less than the corresponding send event.

A path from node n_j^k to node n_i^l represents a *causal chain* of events starting at event e_j^k and ending at event e_i^l . Multiple paths may exist to node e_i^l . For each path, a *cumulative value*, cv , is computed as the sum of each arc label on the path. The length of the path is the number of arcs on the path.

Referring again to figure 3, we see that there are two paths from node n_1^0 to node n_1^3 . The first path, through nodes n_1^0 , n_0^1 , n_0^2 , n_1^2 , and n_1^3 , traverses the arcs connecting those nodes. The *cumulative value* of the path from node n_1^0 to node n_1^3 through the given nodes is the sum of the values of each traversed arc

$$cv(n_1^0, n_0^1, n_0^2, n_1^2, n_1^3) = v_{1,0}^{0,1} + v_{0,0}^{1,2} + v_{0,1}^{2,2} + v_{1,1}^{2,3}.$$

The second path, through nodes n_1^0 , n_1^1 , n_1^2 , and n_1^3 traverses the arcs connecting these nodes. The *cumulative value* of the path from node n_1^0 to node n_1^3 through these nodes is

$$v(n_1^0, n_1^1, n_1^2, n_1^3) = v_{1,1}^{0,1} + v_{1,1}^{1,2} + v_{1,1}^{2,3}.$$

Lemma 1 *Any path, q , through G of length 2 has a cumulative value no less than $\Delta \times decay$.*

Proof:

Let (n, n') and (n', n'') be the arcs in path q . There are two cases to consider. Arc (n, n') is either an intraprocess arc or an interprocess arc.

Case 1 (n, n') is an intraprocess arc
From Property 1 we know that the value of arc (n, n') is no less than $\Delta \times decay$. The value of (n', n'') is either zero, if the arc is interprocess, or at least $\Delta \times decay$, if the arc is intraprocess. Therefore, $cv(q) \geq (\Delta \times decay + 0)$.

Case 2 (n, n') is an interprocess arc
Node n' must represent a message receipt. By Assumption 1, a receipt must be succeeded only by an event on the same process as the receive event. Therefore,

arc (n', n'') must be an intraprocess arc. Hence, $cv(q) \geq (0 + \Delta \times decay)$.

We can conclude that a path, q , through G of length 2 has a cumulative value no less than $\Delta \times decay$. That is, $cv(q) \geq \Delta \times decay$. ■

We now apply Lemma 1 to a path of arbitrary length greater than one. We partition the path into segments of length two and apply Lemma 1 to each segment. By summing the values of the segments, we arrive at a minimum cumulative value for the path.

Partitioning an odd length path results in a single arc that is not part of a two arc segment. In this case, we assume that the final arc is interprocess and has a value of zero. No contribution is made by the final arc towards the cumulative value of the path. In the following lemma, we assume that the length of the path is L which is even. A path of odd length, $L + 1$, is composed of an even length path, L , and the final, interprocess arc which may not contribute to the value of the path.

Lemma 2 *Any path through G of length L must have a cumulative value of at least $\frac{L}{2}(\Delta \times decay)$.*

Proof:

Let Q be a path through G of length L . Let $q_1, q_2, \dots, q_{\frac{L}{2}}$ be a partition of path Q such that the length of each q_i is 2. By Lemma 1, we know that the cumulative value of each q_i is at least $(\Delta \times decay)$. Therefore, the cumulative value of path Q is

$$\begin{aligned} cv(Q) &= cv(q_1) + cv(q_2) + \dots + cv(q_{\frac{L}{2}}) \\ &\geq (\Delta \times decay) + (\Delta \times decay) + \\ &\quad \dots + (\Delta \times decay) \\ &\geq \frac{L}{2}(\Delta \times decay) \end{aligned}$$

Therefore, the minimum cumulative value of a path Q of length L is at least $\frac{L}{2}(\Delta \times decay)$. That is, $cv(Q) \geq \frac{L}{2}(\Delta \times decay)$ if $|Q| = L$. ■

The term *minimum cumulative value* refers to this lower bound of $\frac{L}{2}(\Delta \times decay)$ for a path Q of length L . Suppose that multiple paths exist between two nodes of the graph. Lemma 2 tells us that the minimum cumulative values for each path is directly related to the length of the path. The shortest path will have the *smallest* minimum cumulative value.

Lemma 3 *The smallest minimum cumulative value of a path from n to n' is the minimum cumulative value of the shortest path from n to n' .*

Proof:

Let Q_0, Q_1, \dots, Q_p be the paths through G from node n to node n' . Let L_0, L_1, \dots, L_p be the lengths of the paths, respectively. Assume that path Q_i is the shortest path with length L_i . From Lemma 2 we know that the minimum cumulative values of the paths are $\frac{L_0}{2}(\Delta \times decay)$, $\frac{L_1}{2}(\Delta \times decay), \dots, \frac{L_p}{2}(\Delta \times decay)$. Therefore, the smallest minimum cumulative value is $\frac{L_i}{2}(\Delta \times decay)$. ■

Recall that events in an interval are assigned the weight of one. We want to show not only that the weight in each process reaches zero but that a bound exists on how long it will take a process to reach zero. To accomplish our goal, we derive the length of the path whose minimum cumulative value is one. Since the minimum cumulative value is the minimum amount the weights are reduced along the path, a minimum cumulative value of one guarantees a weight of zero is reached.

Lemma 4 *The minimum cumulative value of a path Q of length L is at least one, $cv(Q) \geq 1$, if $L \geq \frac{2}{(\Delta \times decay)}$.*

Proof:

Let Q be a path of length L such that $L \geq \frac{2}{(\Delta \times decay)}$. From Lemma 2, we know that the minimum cumulative value of Q is $\frac{L}{2}(\Delta \times decay)$. Substituting $\frac{2}{(\Delta \times decay)}$ for L , we reach the following conclusion:

$$\begin{aligned} cv(Q) &\geq \frac{L}{2}(\Delta \times decay) \\ &\geq \frac{2}{\Delta \times decay}(\Delta \times decay) \\ &\geq 1 \end{aligned}$$

Therefore, $cv(Q) \geq 1$ if $L \geq \frac{2}{(\Delta \times decay)}$. ■

We can now show that the weights assigned to events must eventually reach zero. Using the shortest path from the interval to an event, we can deduce the minimum weight reduction that the path must represent. When the path reaches a length such that the cumulative value of the path is at least one, then the weight of the event must be zero.

Theorem 1 *The weight of event e_i^k is zero if the shortest path from any node representing an event in the interval to n_i^k is of length $L \geq \frac{2}{(\Delta \times decay)}$.*

Proof:

Let e_i^k be an event following the end of the interval, I . The weight of e_i^k is computed as one minus the cumulative value of the shortest path from a node representing an event in I to n_i^k . If the cumulative value is greater than 1, then a weight of zero is assigned.

By assumption, the shortest path from any n_j^l , where $e_j^l \in I$, to n_i^k is of length $L \geq \frac{2}{(\Delta \times decay)}$. By Lemma 4, the cumulative value of the path from n_j^l to n_i^k is at least 1. Therefore, the weight of e_i^k is zero. ■

We have shown that the weights assigned to events from a single interval will eventually diminish to zero as the temporal distance from the interval reaches a length computed from the user-defined value *decayTime*. Succeeding intervals will return the weights to their initial value of one. This may occur before the weights of the preceding interval have dissipated. The theorem still holds if we replace the phrase “in the interval” with “in any interval” to allow multiple intervals to effect a single event. We see that only the most recent interval is relevant to the computation of the event’s weight.

4 Conclusions

Locating the source code responsible for implementing a particular software feature is problematic when the exact start and stop times of the feature activation are not known. Many features initiate actions that propagate through the system even after a feature is thought to have completed. Capturing the closeness of events to a feature enables a software engineer to locate the code components that are potentially part of the feature’s implementation.

One method of extracting meaningful information from program traces with imprecise intervals uses weights to identify the relevance of each event to the feature. During the time the feature is known to be active, events are assigned a weight of 1. When the feature is thought to have terminated, the weights are reduced as temporal proximity to the interval increases. Events that closely follow the interval are assigned weights near one,

and events occurring long after the interval are assigned weights of zero. Event weights quantitate the causal affects of the interval.

A general system model has been presented to describe the assignment of weights to events. The model, based on Lamport's formalization of causal order, uses a user-defined value, *decayTime*, to indicate the amount of execution time the effects of the feature should linger. After *decayTime*, the impact of the feature diminishes to a negligible amount.

Weight calculation is based on these factors:

location whether the event occurs while the feature is known to be active or after the feature is thought to have terminated,

type whether the event is the receipt of a message or not, and

time the elapsed time from the previous event of the same process.

Note that the event execution times are localized and therefore do not require synchronization. A proof has been presented to show that the assignment of weights to events corresponds to the software engineers notion of *decayTime* and eventually decreases to zero. Two assumptions are made. First, no single event can simultaneously transmit and receive a message. Second, some measurable amount of time must elapse between successive events on a processor.

Since interprocess communication may experience relatively long propagation delays that do not alter the relevance of the message, the weights of receive events are not decayed as are other events. Instead, it is possible that a receive event may be assigned a weight greater than that of the preceding event on the same processor.

The proof establishes that any event for which the shortest causal path from the end of the interval to that event is at least length L must have a weight of zero. A value for L is computed based on the assumptions and user-defined value. The proof shows that the weight function accurately characterizes the software engineer's intuitive definition of *decayTime*. It also shows that weight assignments indicate the importance of events to the feature and can be used to quantify the causal impact of one event on another. Case studies demonstrate the practical application of event weights in [4]

References

- [1] 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.
- [2] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.
- [3] Sharon Simmons, Dennis Edwards, and Norman Wilde. Locating features in distributed systems. In *Proceedings of The 2004 Conference on Software Engineering Research and Practice*, Las Vegas, Nevada, 2004.
- [4] Sharon Simmons, Dennis Edwards, and Norman Wilde. An approach to feature location in distributed systems. *Journal of Systems and Software*, 2005. In Press, Corrected Proof, Available online 20 January.
- [5] Norman Wilde, Michelle Buckellew, Henry Page, and Vaclav Rajlich. A case study of feature location in unstructured legacy fortran code. In *Proceedings of the Fifth European Conference on Software Maintenance and Reengineering - CSMR'01*, pages 68–76. IEEE Computer Society, March 2001.
- [6] Norman Wilde, Christopher Casey, Joe Vandeville, Gary Trio, and Dick Hotz. Reverse engineering of software threads: A design recovery technique for large multi-process systems. *Journal of Systems and Software*, 43:11–17, 1998.
- [7] Norman Wilde, Juan A. Gomez, Thomas Gust, and Douglas Strasburg. Locating user functionality in old code. In *Proceedings of the IEEE Conference on Software Maintenance*, pages 200–205, Orlando, Florida, November 1992.
- [8] Norman Wilde and Michael Scully. Software reconnaissance: Mapping program features to code. *Journal of Software Maintenance: Research and Practice*, 7:49–62, January 1995.
- [9] W. Eric Wong, Joseph R. Horgan, Swapna S. Gokhale, and Kishor S. Trivedi. Locating program features using execution slices. In *1999 IEEE Symposium on Application-Specific Systems and Software Engineering and Technology*, page 194, March 1999.