Algorithms for finding the causal distributed breakpoint

Toshimitsu Masuzawa[1] Nobuki Tokura[2]

[1]Graduate School of Information Science
Nara Institute of Sciences and Technology
Ikoma, Nara 630-01, Japan
e-mail: masuzawa@is.aist-nara.ac.jp

[2]Faculty of Engineering Science, Osaka University
Toyonaka, Osaka 560 Japan

Abstract

A *causal distributed breakpoint* is one of the fundamental mechanisms for debugging distributed programs. It is initiated by a sequential breakpoint in one process of a distributed computation, and restores each process to the *earliest* state that reflects all events that happened causally before the sequential breakpoint.

This paper presents an algorithm for finding the causal distributed breakpoint when a sequential breakpoint occurs. To find the causal distributed breakpoint efficiently, some information about dependency of events is piggybacked in every message and is logged at each process. The algorithm requires $O(1)$ information in each message and finds the causal distributed breakpoint in $O(n \log n + m)$ time, where n denotes the number of processes and m denotes the number of distinct pairs of processes directly communicating each other. This paper also considers the distributed implementation of the algorithm.

Keywords: distributed computation, breakpoint, debugger, consistent global state, causal relation

1 Introduction

A breakpoint is one of the fundamental mechanisms for debugging programs. At a breakpoint, execution of a program is stopped and the programmer can see the program state. In a sequential program, this breakpoint is sufficient for investigating the program since the program state reflects *exactly* all events that happened before the breakpoint. In a distributed program, however, it is more complicated since events are only partially ordered by causal relation. Even if we can instantaneously stop execution of all processes on the occurrence of a sequential breakpoint in one process, the global state does *not* reflect *exactly* all events that happened causally before the breakpoint.

A *causal distributed breakpoint* (abbreviated as CDB) is initiated by a sequential breakpoint in one process of a distributed computation. It restores each process to the *earliest* state that reflects all events that happened causally before the sequential breakpoint, according to Lamport's partial order of events [9]. In other words, the CDB is the global state reflecting *exactly* all events that happened causally before the sequential breakpoint.

Fowler and Zwaenepoel [4] first introduced the CDB and showed its importance to debugging of distributed programs (referred to as *application programs*). They also presented an algorithm for finding the CDB when a sequential breakpoint occurs in one of the processes. For reaching the CDB, each process needs to roll back to the desired state. To find the rollback point of each process, certain additional information is appended to each message (called *application message*) of the application program, and it is logged at the receiver for each message. The Fowler's algorithm adopts the *event index*, a sequence number of an event within each process, as the extra information. Its worst case time complexity is linear in the number of messages exchanged by the application program [4], while it appends only $O(1)$ extra information to each application message. The reason for such long running time is the *domino effect*, that is, the rollback of one process causes another's rollback, and it also causes another's rollback, and so on.

Aoyagi and Manabe [1] presented an algorithm that finds the CDB in only $O(n)$ time, where n is the number of processes in the application program. It is (asymptotically) time optimal since the trivial lower bound of the running time is $\Omega(n)$. To avoid the domino effect, the algorithm utilizes a (modified) *vector clock* [11] as the extra information, that is, it appends $O(n)$ extra information to each application message.

This paper shows that the CDB can be found in $O(n \log n + m)$ time by appending only $O(1)$ extra information to each application message, where m is the number of distinct pairs of processes that directly communicate each other in execution of the application program. This improves drastically the running time of Fowler's algorithm without increasing the amount of the extra information appended to each message. The extra information used in the algorithm is similar to the logical clock of [9]. Similar technique is utilized in [8] for recovering from processor failures efficiently.

The algorithm described above assumes that a process network is fully connected, that is, each process can directly communicate with any other process. More precisely, the algorithm assumes that there exists a process, called a *coordinator*, in addition to processes appearing in the application program, and that the coordinator can directly communicate with any process in the application program. This paper also considers implementation of the algorithm on any network where each process can communicate only with its neighbors.

2 Distributed System Model

Definitions of this section follow those in [2].

This paper considers a distributed computation carried out by a collection of n deterministic processes, $P_1, P_2, \ldots, P_n$. Processes communicate each other only by exchanging messages through communication channels.

Each process is a state machine, and changes its state only when some event occurs. An event is either an *internal* event, a *send* event, or a *receive* event. An internal event causes only a local state change. A send event causes sending of a message and a local state change. A receive event causes receipt of a message and a local state change that depends on the contents of the received message. A send event and a receive event are said to *correspond* if the message that was sent in the send event is received in the receive event. Channels are assumed to have infinite buffers, to be error-free, and to deliver messages in the order sent. We assume the distributed system is an asynchronous system: there exist no bounds on the relative speeds of processes or on message delays.

Consider a distributed computation. The *local history* of each process P_i during the computation is a (possibly infinite) sequence of events $h_i = e_i^1 e_i^2 e_i^3 \ldots$. In this case, e_i^1 is

the first event executed, e_i^2 is the second event executed, and so on. Let $h_i^k = e_i^1 e_i^2 \ldots e_i^k$ denote an initial prefix of local history h_i containing the first k events. We define h_i^0 to be the empty sequence. Even if the same event occurs several times in the distributed computation, we want to distinguish them. Thus we assume, without loss of generality, that every event is unique: the ID of the process and the sequence number is assigned to every event. Since all events have the unique labels, we sometimes treat a local history as a set rather than a sequence: h_i as a set contains exactly the same events as h_i as a sequence. A distributed computation is informally described by the *global history* $H = h_1 \cup h_2 \cup \cdots \cup h_n$, a set containing all of its events.

Events in global history H are partially ordered by *causal relation*, denoted by $\rightarrow_H$, that is identical to Lamport's *happened before* relation [9]. The relation is defined as follows.

Definition 1 Let H be a global history of a distributed computation. The *causal relation*, denoted by $\rightarrow_H$, on the events in H is defined as follows:

(1) The relation $e_i^k \rightarrow_H e_i^l$ holds if $k < l$, that is, e_i^k occurs before e_i^l at the same process P_i.

(2) The relation $e_i^k \rightarrow_H e_j^l$ holds if e_i^k is a send event and e_j^l is the corresponding receive event.

(3) The relation $\rightarrow_H$ is the smallest transitive relation satisfying the above two conditions.

■

Notice that the relation $\rightarrow_H$ is not reflexive, that is, $e \rightarrow_H e$ does not hold.

With such a model it is common to define a *global state* of a distributed computation as a vector of process states, one for each process. We denote a global state by a tuple $S = (s_1, s_2, \ldots, s_n)$ of process states where s_i represents the state of P_i. A *cut* of a distributed computation is a subset C of its global history H such that C contains an (possibly empty) initial prefix of each of the local histories. We can specify such a cut $C = h_1^{c_1} \cup h_2^{c_2} \cup \cdots \cup h_n^{c_n}$ through the tuple of non-negative integers $(c_1, c_2, \ldots, c_n)$. Let s_i^k denote the local state of process p_i immediately after the execution of event e_i^k and let s_i^0 be

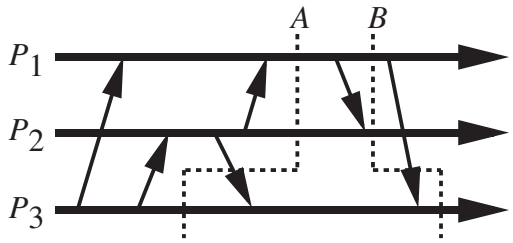

Figure 1: Consistent (A) and inconsistent (B) cuts

its initial state before any events are executed. Clearly, there is a natural correspondence between a cut $C = (c_1, c_2, \ldots, c_n)$ and a global state $S[C] = (s_1^{c_1}, s_2^{c_2}, \ldots, s_n^{c_n})$.

A cut C of a global history H is *consistent*, if for all events e and e' in H,

$$(e \in C) \wedge (e' \rightarrow_H e) \Rightarrow e' \in C$$

holds. We also define a *consistent global state* (CGS) to be the one corresponding to a consistent cut. It is common to depict a distributed computation using a *space-time diagram*. Figure 1 is an example of the diagram with time going from left to right. Message communication is depicted by an arrow from sender's time to receiver's. A cut has a natural graphical interpretation as a partitioning of the space-time diagram along the time axis. Figure 1 illustrates two cuts described by dashed line A and dashed line B. It is easy to verify the consistency of a cut using the diagram: if all arrows that intersect the cut have their bases to the left and heads to the right of it, then the cut is consistent. In Fig. 1 cut A is consistent, while cut B is inconsistent.

3 Causal Distributed Breakpoint

This section gives definition of a causal distributed breakpoint [4] and a brief description of a distributed debugger considered in this paper. The debugger is substantially the same as those proposed in the literatures [7, 10].

The target program of the debugger is a distributed program called an *application program*, where n processes $P_1, P_2, \ldots, P_n$ are engaging in the distributed computation. The debugger has one process P_0, called a *coordinator*, in addition to processes P_i ($1 \leq i \leq n$). The coordinator P_0 has a bidirectional communication channel to each process

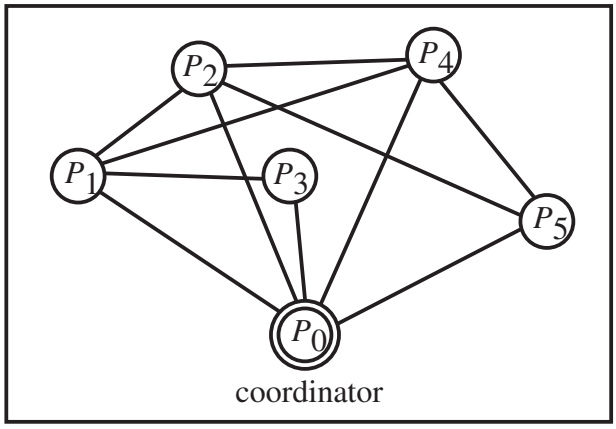

Figure 2: Debugger model for distributed computing

P_i $(1 \leq i \leq n)$ as illustrated in Fig. 2. In using the debugger, the user can set *breakpoints* by specifying predicates. We assume that a predicate is restricted to an expression for one process: it can be determined solely from the process state whether the predicate is true or false. We say that a breakpoint *occurs* at process P_i when a breakpoint predicate concerning P_i becomes true.

Consider a breakpoint occurs at some process P_B $(1 \leq B \leq n)$ when P_B executes an event e_B. The process P_B is called a *breakpoint process*, and the event e_B is called a *breakpoint event*. The occurrence of the breakpoint is detected by P_B and notified to the coordinator P_0. Then, the coordinator P_0 stops the execution of all processes. The debugger also has a facility for rollbacking the other processes than P_B to restore the consistent global state corresponding to a cut called the *causal distributed breakpoint*. Intuitively, the causal distributed breakpoint is the minimal consistent cut including e_B.

Definition 2 Let H be any global history of a distributed computation, and e be any event in H. The cut $C = (c_1, c_2, \ldots, c_n)$ is the *causal distributed breakpoint* (CDB) induced from e, if the following condition holds for each i $(1 \leq i \leq n)$.

(a) If $e = e_i^j$ (i.e., e is the j^{th} event of P_i), then $c_i = j$.

(b) If e is not an event of P_i and there is no event e' of P_i such that $e' \rightarrow_H e$, then $c_i = 0$.

(c) If e is not an event of P_i and there is an event e' of P_i such that $e' \rightarrow_H e$, then $c_i = \max\{k \mid e_i^k \rightarrow_H e\}$.

For convenience, a fictive event e_i^0 is introduced, and $e_i^{c_i}$ is called the *CDB event* of P_i. The process state $s_i^{c_i}$ is also called the *CDB state* of P_i. ■

We define the CDB as a cut, while it is defined as a global state in [1, 4]. The CDB defined as a global state is the global state corresponding to the CDB as a cut. Thus, there is no practical difference between them.

The following lemma can be easily proved from the definition of the CDB.

Lemma 1 Let H be any global history of a distributed computation, e be an arbitrary event in H, and $C = (c_1, c_2, \ldots, c_n)$ be the CDB induced from e. The followings hold.

(1) The CDB C is a consistent cut.

(2) Assume e be an event of P_i. For any j $(1 \leq j \leq n, j \neq i)$, $e_j^{c_j}$ is a send event if $c_j \geq 1$.

■

In order to restore each process to the CDB state efficiently, the checkpointing activity also proceeds in all processes: each process occasionally saves its process state. Each process restores its CDB state by restoring a prior checkpoint state and then replaying its original execution to the CDB state. Execution of a distributed program is, however, nondeterministic because of the asynchrony of processes and communications. Hence, each process records its event order when an application program is executed, and replays the original execution by means of the event order logs.

Consequently the debugger has two modes, *recording* and *restoring* modes. It starts execution of an application program in the recording mode, and transfers to the restoring mode on occurrence of a breakpoint in some process.

(A) *Recording mode:* each process P_i $(1 \leq i \leq n)$ normally executes an application program, records its event order logs, and occasionally saves its process state.

(B) *Restoring mode:* the CDB state is restored by replaying the original execution from a prior checkpoint state.

This paper considers only the problem for determining the CDB when a breakpoint occurs. Checkpointing and replaying are beyond the scope of this paper.

4 Algorithm for Finding the CDB

4.1 Send Event Index

Let P_B be the breakpoint process, e_B $(= e_B^{c_B})$ be the breakpoint event, and $C = (c_1, c_2, \ldots, c_n)$ be the CDB induced from e_B. It follows from Lemma 1 that $e_i^{c_i}$ $(i \neq B)$ is a send event if $c_i \geq 1$. Thus, a positive integer $I(e)$, called the *send event index* (SEI), is assigned to every send event e. The SEI is unique within a process, that is, $I(e) \neq I(e')$ holds for any distinct send events e and e' of the same process. Thus, the algorithm uses the SEI for specifying every CDB event of process P_i $(i \neq B)$.

The SEI is a slight modification of Lamport's logical clock [9]: while Lamport's logical clock assigns a clock value to every event, the SEI is assigned only to every send event.

Definition 3 Let H be a global history of a distributed computation. We assign a *send event index* (denoted by $I(e)$) to every send event e in H as follows.

(a) If there is a send event e' such that $e' \rightarrow_H e$, then
$I(e) = \max\{I(e') \mid e' \text{ is a send event such that } e' \rightarrow_H e\} + 1$.

(b) If there is no send event e' such that $e' \rightarrow_H e$, then $I(e) = 1$.

■

Figure 3 shows an example of the SEI where each send event is annotated with its SEI.

We introduce a *CDB index* to specify the CDB. For convenience, we assume $I(e_i^0) = 0$ for each fictive event e_i^0.

Definition 4 Let H be the global history of a distributed computation, $e_B (= e_B^{c_B})$ be the breakpoint event, and $C = (c_1, c_2, \ldots, c_n)$ be the CDB induced from e_B. The *CDB index* of each process P_i $(i \neq B)$ is the SEI assigned to the CDB event of P_i, that is, $I(e_i^{c_i})$. ■

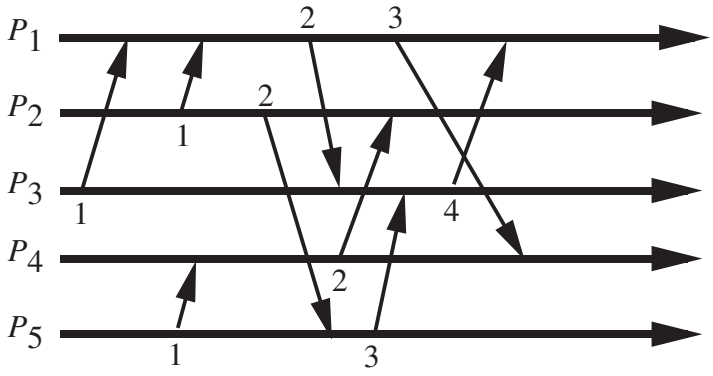

Figure 3: Send event indices

4.2 Algorithm for Recording Mode

In order to assign the SEI to every send event, each application message carries with it the SEI of the corresponding send event. Every process P_i $(1 \leq i \leq n)$ has vector RI_i of size n, called a *received index vector*. Its j^{th} $(1 \leq j \leq n, j \neq i)$ component, $RI_i[j]$, stores the largest SEI P_i has received from P_j, and the i^{th} component, $RI_i[i]$, stores the SEI of the latest send event of P_i. At the beginning of execution of the application program, every component of RI_i stores 0 as the initial value. Note that the component $RI_i[j]$ $(j \neq i)$ remains 0 until P_i receives an application message from P_j, and $RI_i[i]$ remains 0 until P_i execute a send event. The value of RI_i at s_i^k $(k \geq 0)$ is referred to as "RI_i of e_i^k".

Figure 4 presents the algorithm for the recording mode. In the algorithm, the SEI is appended to every application message, and RI_i of every send event is recorded in its log. Indentation indicates block structures in the figure.

4.3 Algorithm for Restoring Mode

This subsection presents the algorithm to find the CDB induced from a given breakpoint event. The algorithm finds the CDB index of every process other than the breakpoint process. In what follows, let P_B be the breakpoint process, e_B $(= e_B^{c_B})$ be the breakpoint event, and $C = (c_1, c_2, \ldots, c_n)$ be the CDB induced from e_B. Precisely, the algorithm finds the $I(e_i^{c_i})$ for each i $(1 \leq i \leq n, i \neq B)$. There is no difficulty in determining c_i or $e_i^{c_i}$ from $I(e_i^{c_i})$, for the SEI is unique within a process. Thus, we consider that the CDB is determined if the CDB index of every process other than P_B is found.

Algorithm for application process P_i $(1 \leq i \leq n)$

(a) Initialization: /* Initialize RI_i. */

for each j $(1 \leq j \leq n)$

$RI_i[j] := 0$;

(b) On sending a message M **to** P_j**:**

$RI_i[i] := \max\{RI_i[j] \mid 1 \leq j \leq n\} + 1$;

send$(P_j, \ll M, RI_i[i] \gg)$; /* Send M with the SEI. */

record(RI_i); /* Log RI_i of the send event. */

(c) On receipt of a message $\ll M, a \gg$ **from** P_j**:**

$RI_i[j] := a$;

process M; /* Process M according to the application program. */

Figure 4: Algorithm for recording mode

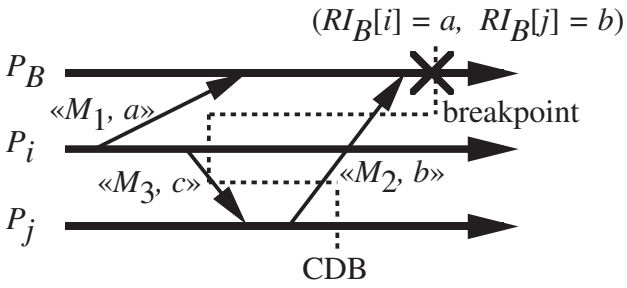

Figure 5: An example of execution

The algorithm is invoked when a breakpoint occurs at P_B. If $RI_B[i] = a$ (≥ 1) holds at the breakpoint, then P_B has received an application message carrying SEI a from P_i. This implies that the CDB index of P_i is not smaller than a. Note that the CDB index of P_i is not necessarily a. For example, consider the computation in Fig. 5. In Fig. 5, $RI_B[i] = a$ and $RI_B[j] = b$ hold at the breakpoint. Before sending message $\ll M_2, b\gg$, P_j received message $\ll M_3, c\gg$ from P_i. Message $\ll M_1, a\gg$ is sent before $\ll M_3, c\gg$, that is, $c > a$ holds. The send event of $\ll M_3, c\gg$ at P_i, thus, occurs causally before the breakpoint. As a consequence, the CDB index of P_i is not a but c.

As described in the above example, we cannot determine the CDB indices solely from RI_B at the breakpoint. The coordinator P_0 coordinates the task of finding the CDB indices. The coordinator P_0 has a vector CI of size n, called *CDB index vector*. Its component $CI[i]$ ($1 \leq i \leq n, i \neq B$) stores the (temporary) CDB index of process P_i, and its initial value is 0.

Each process is *marked* or *unmarked*: every process is initially unmarked, becomes marked when its CDB index is determined, and remains marked until the algorithm terminates.

On occurrence of the breakpoint, the breakpoint process P_B sends a message $\ll$**START**, $RI'_B\gg$ to P_0 where RI'_B is (the value of) RI_B of e_B. On receiving the message, P_0 sets CI to RI'_B, and then updates $CI[B]$ to a special symbol '*' to indicate that P_B is the breakpoint process. The breakpoint process becomes a marked process.

The main part of the algorithm for finding the CDB index of every process other than P_B consists of $n-1$ iterations. In the k^{th} iteration ($1 \leq k \leq n-1$), P_0 finds the process $P_{\varphi(k)}$ such that $CI[\varphi(k)]$ is largest among $\{CI[j] \mid P_j$ is unmarked$\}$, and then

the CDB index of $P_{\varphi(k)}$ is determined to be $CI[\varphi(k)]$ and $P_{\varphi(k)}$ becomes marked. If no process remains unmarked, then the algorithm terminates. Otherwise, the coordinator P_0 sends a message $\ll$**CDB**, $CI[\varphi(k)]\gg$ to $P_{\varphi(k)}$ in order to notify $P_{\varphi(k)}$ of its CDB index. On receiving the message, $P_{\varphi(k)}$ finds $RI_{\varphi(k)}$ of the CDB event from its log, and sends a message $\ll$**REPORT**, $RI'_{\varphi(k)}\gg$ to P_0, where $RI'_{\varphi(k)}$ is $RI_{\varphi(k)}$ of the CDB event. When P_0 receives the message, it updates CI to $\max\{CI, RI'_{\varphi(k)}\}$ (the componentwise maximum) and proceeds to the next iteration. Figure 6 presents the algorithm.

(Remark) We can easily reduce the number of iterations in some cases by the following modification:

(a) The k^{th} iteration simultaneously determines the CDB indices of all processes $P_{\varphi(k)}$ such that $CI[\varphi(k)] = \max\{CI[j] \mid P_j \text{ is unmarked}\}$.

(b) The algorithm terminates immediately when $\max\{CI[j] \mid P_j \text{ is unmarked}\} = 0$ holds.

In the worst case, however, the modified algorithm also requires $n-1$ iterations, and, for simplicity, this paper does not adopt the modified version. ■

4.4 Example

We show an example of execution of the algorithm. Figure 7 shows an execution of an application program, where every send event is annotated with its SEI. Assume that a breakpoint occurs at process P_1. Since $RI_1 = (3, 1, 4, 0, 0)$ at the breakpoint, P_0 sets CI to $(*, 1, 4, 0, 0)$ and P_1 becomes a marked process. In the first iteration, the CDB index of P_3 is determined to be 4, and it is notified to P_3. Process P_3 also becomes a marked process. Since RI_3 of the CDB event is $(2, 0, 4, 0, 3)$, CI is updated to $(*, 1, 4, 0, 3)$. In the second iteration, the CDB index of P_5 is determined to be 3, and P_5 becomes a marked process. Since RI_5 of the CDB event is $(0, 2, 0, 0, 3)$, CI is updated to $(*, 2, 4, 0, 3)$. The algorithm determines the CDB indices of P_2 and P_4 in the third and the fourth iterations respectively. At the end of the algorithm, CI stores $(*, 2, 4, 0, 3)$ that are the CDB indices of all processes other than the breakpoint process.

(A) Algorithm for application process P_i $(1 \leq i \leq n)$

(a) On occurrence of the breakpoint:

/* P_i is the breakpoint process. */

$RI'_i := RI_i$ of the breakpoint event;

send(P_0, $\ll$**START**, $RI'_i \gg$);

(b) On receipt of $\ll$CDB, $c\gg$ from P_0:

$RI'_i := RI_i$ of the send event e s.t. $I(e) = c$;

send(P_0, $\ll$**REPORT**, $RI'_i \gg$);

(B) Algorithm for the coordinator P_0

/* Initialized on receipt of $\ll$**START**, $RI'_B \gg$ from P_B */

$CI := RI'_B$; $CI[B] :=' *'$;

$UnMarked := \{i \mid 1 \leq i \leq n, i \neq B\}$;

while $UnMarked \neq \emptyset$

$k := j(\in UnMarked)$ s.t. $CI[j] = \max\{CI[l] \mid l \in UnMarked\}$;

$UnMarked := UnMarked - \{k\}$;

send(P_k, $\ll$**CDB**, $CI[k] \gg$);

wait until receipt of $\ll$**REPORT**, $RI'_k \gg$ from P_k;

for each l $(\in Unmarked)$

$CI[l] := \max\{CI[l], RI'_k[l]\}$;

Figure 6: Algorithm for restoring mode

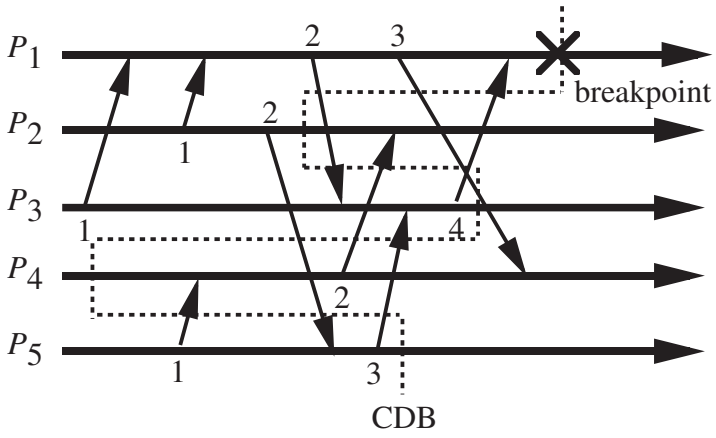

Figure 7: An example of CDB finding

4.5 Correctness

In this section, we show that the algorithm determines the CDB correctly.

Lemma 2 Let P_i $(1 \leq i \leq n)$ be any process and e_i be any event of P_i. If $RI_i[j] = r \geq 1$ $(j \neq i)$ holds for RI_i of e_i (i.e. RI_i immediately after e_i), then $e_j \rightarrow_H e_i$ holds where e_j is the send event of P_j such that $I(e_j) = r$.

(Proof) The fact $RI_i[j] = r$ implies that P_i has received a message from P_j that carries the SEI r. The lemma clearly holds. ■

Lemma 3 Let P_i $(1 \leq i \leq n)$ be any process and e_i be any send event of P_i. Letting $RI_i' = (r_1, r_2, \ldots, r_n)$ be (the value of) RI_i of e_i, $r_j < r_i$ holds for any j $(1 \leq j \leq n, j \neq i)$.

(Proof) The value r_i is the SEI of e_i. From the definition of the SEI, the lemma obviously holds. ■

Lemma 4 Let CI^F be the value of the variable CI when the algorithm terminates. If $CI^F[i] = c \geq 1$ $(i \neq B)$, then $e_i \rightarrow_H e_B$ holds, where e_i is the send event of P_i such that $I(e_i) = c$.

(Proof) Let CI^k $(1 \leq k \leq n)$ be the value of CI at the beginning of the k^{th} iteration if $k \leq n-1$, and $CI^n = CI^F$. By induction on k, we show the following claim.

Claim: If $CI^k[i] = c \geq 1$ $(i \neq B)$, then $e_i \rightarrow_H e_B$ holds, where e_i is the send event of P_i such that $I(e_i) = c$.

At the beginning of the first iteration (i.e. $k = 1$), $CI^1[i] = RI'_B[i]$ holds where RI'_B is RI_B at the breakpoint. From Lemma 2, the claim holds.

Assuming that the claim holds at the beginning of the k^{th} iteration ($1 \leq k \leq n-1$), we show that the claim holds at the beginning of the $k+1^{st}$ iteration (i.e. at the end of the k^{th} iteration). In the k^{th} iteration, an unmarked process, say $P_{\varphi(k)}$, becomes marked, and CI is updated using $RI_{\varphi(k)}$ of the send event $e_{\varphi(k)}$ such that $I(e_{\varphi(k)}) = CI^k[\varphi(k)]$. If $CI[i]$ does not change in the k^{th} iteration, it is clear from the inductive hypothesis that the claim holds at the beginning of the $k+1^{st}$ iteration. Consider the case that $CI[i]$ changes, i.e., $CI^{k+1}[i] = RI'_{\varphi(k)}[i]$ holds where $RI'_{\varphi(k)}$ is $RI_{\varphi(k)}$ of $e_{\varphi(k)}$. From Lemma 2, $e_i \rightarrow_H e_{\varphi(k)}$ holds. From the inductive hypothesis, $e_{\varphi(k)} \rightarrow_H e_B$ holds. Therefore, the claim holds at the beginning of the $k+1^{st}$ iteration. ■

Lemma 5 Let CI^F be the value of the variable CI when the algorithm terminates. For any k and k' ($k < k'$), let $P_{\varphi(k)}$ and $P_{\varphi(k')}$ be the processes that become marked in the k^{th} and the k'^{th} iterations respectively. Then, $CI^F[\varphi(k)] \geq CI^F[\varphi(k')]$ holds.

(Proof) Since $P_{\varphi(k)}$ becomes marked in the k^{th} iteration, $CI^k[\varphi(k)] = \max\{CI^k[j] \mid P_j$ is unmarked at the beginning of the k^{th} iteration$\}$ holds, where CI^k is CI at the beginning of the k^{th} iteration. It follows that $CI^k[\varphi(k)] \geq CI^k[\varphi(k')]$. In each iteration p ($k \leq p < k'$), $CI[\varphi(k')]$ may change, however, $CI^F[\varphi(k')] \leq CI^k[\varphi(k)]$ holds from Lemma 3. Since $CI[\varphi(k)]$ does not change in or after the k^{th} iteration, the lemma holds. ■

Theorem 1 The algorithm determines the CDB correctly, when given a breakpoint event.

(Proof) Without loss of generality, we assume, in this proof, the breakpoint process and the breakpoint event are P_n and $e_n (= e_n^{c_n})$, respectively. Let $C = (c_1, c_2, \ldots, c_n)$ be the (true) CDB induced from e_n, and $C' = (c'_1, c'_2, \ldots, c'_n)$ be the cut determined as the CDB by the algorithm. That is, the cut C' is determined as follows, where CI^F is the value of variable CI when the algorithm terminates.

(a) $c'_n = c_n$.

(b) For each i ($1 \leq i \leq n-1$), $c'_i = 0$ if $CI^F[i] = 0$.

(c) For each i ($1 \leq i \leq n-1$), $e_i^{c'_i}$ is the send event such that $I(e_i^{c'_i}) = CI^F[i]$ if $CI^F[i] \geq 1$.

We prove $C' = C$. Since $c'_n = c_n$ from the definition of C', it is sufficient to show $c'_i = c_i$ for each i $(1 \leq i \leq n-1)$. From the definition of the CDB and Lemma 4, $c'_i \leq c_i$ holds. In the rest of this proof, we show $c'_i \geq c_i$.

For contradiction, assume that there exists j $(1 \leq j \leq n-1)$ such that $c'_j < c_j$. Let j be such that $c'_j < c_j$ and $I(e_j^{c_j}) = \max\{I(e_k^{c_k}) \mid c'_k < c_k\}$. From the definition of the CDB, $e_j^{c_j} \rightarrow_H e_n$ holds and $e_j^{c_j}$ is a send event.

If a message sent by P_j at or after $e_j^{c_j}$ is received by P_n at or before e_n, $CI^F[j] \geq RI'_n[j] \geq I(e_j^{c_j})$ holds where RI'_n is RI_n of e_n. That contradicts $c'_j < c_j$.

Otherwise, there exists a send event e_l of P_l $(1 \leq l \leq n-1, l \neq j)$ such that $e_j^{c_j} \rightarrow_H e_l \rightarrow_H e_n$ and such that a message sent by P_j at or after $e_j^{c_j}$ is received by P_l before e_l. This implies that $I(e_l) > I(e_j^{c_j})$. From $e_l \rightarrow_H e_n$ and the definition of the CDB, $I(e_l^{c_l}) \geq I(e_l)$ holds. Thus, $I(e_l^{c_l}) > I(e_j^{c_j})$. It follows from the definition of j that $c'_l \geq c_l$ holds, i.e., $CI^F[l] \geq I(e_l^{c_l})$. Therefore, $CI^F[l] \geq I(e_l^{c_l}) > I(e_j^{c_j}) > CI^F[j]$. This implies from Lemma 5 that P_l becomes marked before P_j. When P_l becomes marked, $CI[j]$ is updated to $\max\{CI[j], RI'_l[j]\}$ where $RI'_l[j]$ is the value of $RI_l[j]$ of $e_l^{c'_l}$. However $RI'_l[j] \geq I[e_j^{c_j}]$ holds since P_l receives a message from P_j before e_l that carries the SEI of $I[e_j^{c_j}]$ or more. That contradicts $c'_j < c_j$. ■

4.6 Complexity

We now show message and time complexities of the algorithm. We consider the complexities under the following assumptions.

Assumption 1 A message can carry a constant number of process IDs and SEIs. ■

Assumption 2 The delay experienced by a message in a channel is bounded by some constant. ■

Notice that we assume the bounded delay (Assumption 2) only for measuring time complexity of the algorithm. The algorithm works correctly without the assumption.

Theorem 2 The algorithm determines the CDB in $O(n \log n + m)$ time and with $O(n+m)$ messages where m is the number of distinct pairs of processes that directly communicate each other in the original execution of the application program.

(Proof) We adopt Fibonacci heap [3] to implement vector CI. In the k^{th} iteration ($1 \leq k \leq n-1$), the coordinator P_0 executes one *Extract-Max* operation to select the unmarked process, say $P_{\varphi(k)}$, with the largest CI. When P_0 receives $\ll$**REPORT**, $RI'_{\varphi(k)} \gg$ from $P_{\varphi(k)}$, it also executes at most $d(P_{\varphi(k)})$ *Increase-Key* operations to update CI, where $d(P_{\varphi(k)})$ represents the number of processes that has sent a message to $P_{\varphi(k)}$ in the original execution of the application program. (Notice that it is sufficient to update $CI[j]$ only if $RI'_{\varphi(k)}[j] \geq 1$.) Therefore, the algorithm overall requires $n-1$ *Extract-Max* operations and at most m *Increase-Key* operations. These operation can be executed in time $O(n \log n + m)$.

When the breakpoint occurs, the breakpoint process P_B sends $\ll$**START**, $RI'_B \gg$ to P_0. Since RI'_B is a vector of size n, it is required that $\ll$**START**, $RI'_B \gg$ is divided into some messages (Assumption 1). Since it is sufficient to send the component $RI'_B[j]$ such that $RI'_B[j] \geq 1$, $\ll$**START**, $RI'_B \gg$ can be sent by $O(d(P_B))$ messages and it takes time $O(d(P_B))$.

In the k^{th} iteration, the coordinator P_0 sends $\ll$**CDB**, $CI[\varphi(k)] \gg$ to $P_{\varphi(k)}$. It requires one message and takes time $O(1)$. When $P_{\varphi(k)}$ receives the message, it sends $\ll$**REPORT**, $RI'_{\varphi(k)} \gg$ to P_0. It also requires $O(d(P_{\varphi(k)}))$ messages and it takes time $O(d(P_{\varphi(k)}))$.

Therefore, the running time is $O(n \log n + m)$ and the total number of messages exchanged is $O(n + m)$ for the overall algorithm. ■

5 Implementation on Networks

This section considers a situation where processes of application program are distributed over an arbitrary network. For simplicity, it is assumed that each processor of the network has exactly one application process. In such a situation, each process can directly send messages only to the processes in the neighboring processors. In the algorithm proposed in the previous section, however, it is assumed that the coordinator can directly communicate with any process of the application program. Thus, the algorithm cannot be adapted to the circumstances considered in this section. This section shows how the algorithm can be modified so that it can be adapted to the situation where processes of the application program are distributed over an arbitrary network.

Table 1: Message complexities of the algorithm with central coordinator

	Number of edges	Diameter	Message Complexity
(a) Hypercube	$(n \log n)/2$	$\log n$	$O(n \log^2 n)$
(b) Cube-connected-cycles	$3n/2$	$\leq 2 \log n$	$O(n \log n)$
(c) Shuffle-exchange graph	$3n/2$	$2 \log n$	$O(n \log n)$
(d) r-dimensional array	$\leq rn$	$r(n^{1/r} - 1)$	$O(r^2 n^{1+1/r})$

5.1 Centralized Coordinator

The algorithm proposed in the previous section can be simply implemented in any network as follows. Before execution of the algorithm, the coordinator P_0 is assigned to some processor of the network and a breadth-first spanning tree (abbreviated as BFST) of the network is constructed. The coordinator communicates with the application processes by means of the BFST in execution of the algorithm.

Theorem 3 The CDB can be found with $O((n+m)D)$ messages in an arbitrary network where n and m are the numbers of processors and communication links in the network respectively and D stands for the diameter of the network.

(Proof) Since every application process communicate with P_0 using the BFST, a single message of the previous section can be transferred to its destination using at most D messages. Therefore, the theorem holds from Theorem 2. ■

Note that the message complexity of Theorem 3 remains $O((n + m)D)$ even if messages used for constructing the BFST is included in the count, since the BFST can be constructed with $O(\sqrt{nDm} + m)$ [12] messages.

The desirable networks for distributed or parallel computation have small numbers of edges and small diameters. In such networks the algorithm works efficiently. The message complexities of the algorithm on some representative networks are summarized in Table 1 (See Appendix for definitions of the networks).

5.2 Decentralized Coordinator

This subsection shows another implementation of the algorithm proposed in Section 4. In the implementation, the coordinator P_0 is implemented in a distributed fashion. In the algorithms described above, the CDB index vector CI is maintained by the coordinator. In the implementation of this subsection, however, CI is distributed over the network. That is, each process P_i maintains $CI[i]$.

Before execution of the algorithm, a rooted spanning tree (abbreviated as ST) of the network is constructed. In order to find the largest element of $\{CI[j] \mid P_j \text{ is unmarked}\}$, tree computation is executed on the ST: each unmarked leaf process P_i sends its parent the value of its $CI[i]$, and each internal process P_i sends its parent the largest element of CI among those its unmarked descendants have, that is, each internal process P_i sends its parent the largest element of $\{CI[j] \mid P_j \text{ is an unmarked descendant of } P_i\}$. When a process, say P_i, becomes a marked process, the $CI[l]$ of every neighboring process, say P_l, has to be updated by $CI[l] := \max\{CI[l], RI'_i[l]\}$ where RI'_i stands for the value of RI_i of its CDB event. Therefore the process P_i has only to send a message carrying $RI'_i[l]$ to its neighboring process P_l.

Theorem 4 The CDB indices of all processes can be found with $O(n^2)$ messages in an arbitrary network where n is the numbers of processors in the network.

(Proof) In order to determine the largest element of $\{CI[j] \mid P_j \text{ is unmarked}\}$, $n-1$ messages are used in the tree computation. Since $n-1$ tree computations are executed in the whole algorithm, it takes $O(n^2)$ messages. It also takes $O(m)$ messages for transferring an element of RI between every neighboring processors. Therefore, it follows from $m \leq n(n-1)/2$ that the CDB indices of all processes can be found with $O(n^2)$ messages in an arbitrary network. ■

Notice that the message complexity of Theorem 4 remains $O(n^2)$ even if messages used for constructing the ST is included in the count, since a ST can be constructed with $O(n \log n + m)$ messages [6].

6 Conclusions

This paper showed an algorithm that finds the causal distributed breakpoint in $O(n \log n + m)$ time by appending only $O(1)$ extra information to each application message. This paper also presented algorithms for finding the causal distributed breakpoint on any network and discussed their message complexities.

In distributed program debugger, it may be required to find the minimum cut containing some given events. The algorithm in this paper can be easily extended so that it can find such a cut with the same complexities.

Acknowledgments The authors thank Dr. Yoshifumi Manabe and Mr. Shigemi Aoyagi (NTT Software Laboratories) for their comments on this work, and thank Ms. Michiko Inoue (Graduate School of Osaka University) for her helpful discussion on the implementation on networks. The first author also thanks Prof. Hideo Fujiwara (Nara Institute of Science and Technology) for his suggestion to his research.

References

[1] S. Aoyagi and Y. Manabe. Checkpoint and rollback algorithm for distributed debugger. Technical Report SW-92-10-4, JSSS, April 1992. (in Japanese).

[2] O. Babaoglu and K. Marzullo. Consistent global states of distributed systems: Fundamental concepts and mechanisms. In S. Mullender, editor, *Distributed Systems (Second edition)*, chapter 4, pages 55 – 96. Addison-Wesley, 1993.

[3] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction To Algorithms*, chapter 21. McGraw-Hill, 1989.

[4] J. Fowler and W. Zwaenepoel. Causal distributed breakpoints. In *Proc. of 10th International Conference on Distributed Computing Systems*, pages 134–141, May 1990.

[5] F.T.Leighton. *Introduction to Parallel Algortihms:Arrays, Trees, Hypercubes.* Morgan Kaufmann Publishers, 1992.

[6] R. G. Gallager, P. A. Humblet, and P. M. Spira. A distributed algorithm for minimum-weight spanning tree. *ACM Transactions on Programming Languages and Systems*, 5(1):66–77, 1991.

[7] A. P. Goldberg, A. Gopal, A. Lowry, and R. Strom. Restoring consistent global states of distributed computations. *ACM Sigplan Notice (Proc. of ACM/ONR Workshop on Parallel and Distributed Debugging (May 1991))*, 26(12):144–154, December 1991.

[8] T. T. Y. Juang and S. Venkatesan. Efficient algorithms for crash recovery in distributed systems. In *Foundations of Software Technology and Theoretical Computer Science (LNCS 472)*, pages 349–361, December 1990.

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

[10] Y. Manabe and M. Imase. Global conditions in debugging distributed programs. *Journal of Parallel and Distributed Computing*, 15(1):62–69, May 1992.

[11] F. Mattern. Efficient algorithms for distributed snapshots and global virtual time approximation. In *Proc. of the International Workshop on Parallel and Distributed Algorithms*, pages 215–226, 1989.

[12] K. Miura, T. Masuzawa, and N. Tokura. A distributed shortest paths algorithm with distance-dependent message complexities. *Transactions of IEICE*, J77-D-I(1):21–32, January 1994. (in Japanese).

A Definitions of Networks

In this Appendix, we define the networks in Table 1. These definitions are taken from [5].

A.1 Hypercube

The r-dimensional hypercube has $n = 2^r$ nodes and $m = r2^{r-1} (= (n \log n)/2)$ edges. Each node corresponds to an r-bit binary string, and two nodes are linked with an edge if and only if their binary string differ in precisely one bit. An edge is called a dimension k edge if it links two nodes that differ in the k^{th} bit position. Its diameter is clearly $r (= \log n)$.

A.2 Cube-Connected-Cycles (CCC)

The r-dimensional cube-connected-cycles is constructed from the r-dimensional hypercube by replacing each node of the hypercube with a cycle of r nodes. The i^{th} dimension edge incident to a node of the hypercube is then connected to the i^{th} node of the corresponding cycle of the CCC. The r-dimensional cube-connected-cycles has $n = r2^r$ nodes and $m = 3r2^{r-1} (= 3n/2)$ edges. More precisely, the r-dimensional cube-connected-cycles is defined as follows. Each node is represented by a pair $\langle w, i \rangle$ where w is any r-bit binary string and i $(1 \leq i \leq r)$. Then two nodes $\langle w, i \rangle$ and $\langle w', i' \rangle$ are linked by an edge in the CCC if and only if either

(1) $w = w'$ and $i = i' \pm 1$ **mod** r, or

(2) $i = i'$ and w differs from w' in precisely the i^{th} bit.

The diameter of the r-dimensional cube-connected-cycles is $O(\log n)$.

A.3 Shuffle-Exchange Graph

The r-dimensional shuffle-exchange graph has $n = 2^r$ nodes and $m = 3r2^{r-1} (= 3n/2)$ edges. Each node corresponds to a unique r-bit binary number, and two nodes u and v are linked by an edge if and only if either

(1) u and v differ in precisely the last bit, or

(2) u is a left or right cyclic shift of v.

The diameter of the r-dimensional shuffle-exchange graph is $2\log n - 1$.

A.4 r-Dimensional Array

The r-dimensional s-sided array has $n = s^r$ nodes and $m = rs^r - rs^{r-1} (\leq rn)$ edges. Each node corresponds to an s-array r-vector $(i_1, i_2, \ldots, i_r)$ where $1 \leq i_j \leq s$ for $1 \leq j \leq r$. Two nodes are linked by an edge if and only if

(1) they differ in precisely one coordinate, and

(2) the absolute value of the difference in that coordinate is 1.

The diameter of the r-dimensional s-sided array is $r(s-1)$. In the special case that $s = 2$, the array is the r-dimensional hypercube.