Optimal Run–Time Tracing of Message–Passing Programs

Anish Karmarkar and Nitin Vaidya

Department of Computer Science, Texas A&M University,

College Station, Tx – 77843.

Sept., 1995[1]

Abstract

The widespread adoption of distributed computing has accentuated the need for an effective set of support tools to facilitate debugging and monitoring of distributed programs. Unfortunately for distributed programs, this is not a trivial task. Distributed programs are inherently non–deterministic in nature. Two runs of the same programs with the same input data do not result in the same execution sequence. Cyclic debugging is one of the most common strategies used in debugging. To allow cyclic debugging, messages are traced for repeatable execution. In this paper we present a simple proof that it is impossible to have an algorithm, which will produce an optimal message trace (least number on messages tracde), in general. We then present two algorithms, Algorithm A and Algorithm B. Both the algorithms trace messages at run–time, i.e., when a message is received at a process. Algorithm A does optimal tracing of messages, given the fact that messages are traced at run–time, and no information about the future is available when these decisions are made. Algorithm B improves on the storage requirement and execution time of Algorithm A, and is based on the observation that only (n–1) buffers are required per process for optimal run–time decision making, where n is the number of processes in the system. This algorithm is an improvement over the algorithm presented in [10], which does optimal tracing only when the races amongst messages are transitive.

---

[1] Originally prepared as a term paper for Distributed Algorithms (CS689) in Spring 1995.

1. Introduction

The widespread adoption of distributed computing has accentuated the need for an effective set of support tools to facilitate debugging and monitoring of distributed programs. Unfortunately for distributed programs, this is not a trivial task. Distributed programs are inherently non–deterministic in nature.

Debugging a single sequential program itself is not a trivial task. The added complexity of debugging concurrent programs makes it even harder. There are several problems in debugging concurrent programs. The biggest problem being non–determinacy. This non–determinacy gives rise to non–repeatability. The same programs when executed on the same input may give different results on different runs. Another important factor that makes analysis of distributed programs difficult is the lack of a synchronized global clock [1]. Without a global clock it may be difficult to determine the precise order of events occurring in distinct concurrently executing processors.

The approach usually used in debugging sequential programs is to execute the program till an error occurs. Then, the same program is re–executed with breakpoints or debug statements placed at strategic points in the program. The program is stopped during execution, its state examined and then continued or re–executed. This method is called cyclic debugging. Distributed programs do not, unfortunately, lend themselves easily to this style of debugging.

Consider a system of three processes 1, 2 and 3 as shown in figure 1. Here the messages m1 and m2 race with each other. Depending on the scheduling and message latencies, m1 can be received by process 2 before m2 as shown in figure 1, or m2 can be received before m1 by process 2 as shown in figure 2. This leads to non–determinacy. In fact, it is possible that if the undesirable behavior occurs with a low probability, the programmer may not be able to reproduce the error situation.

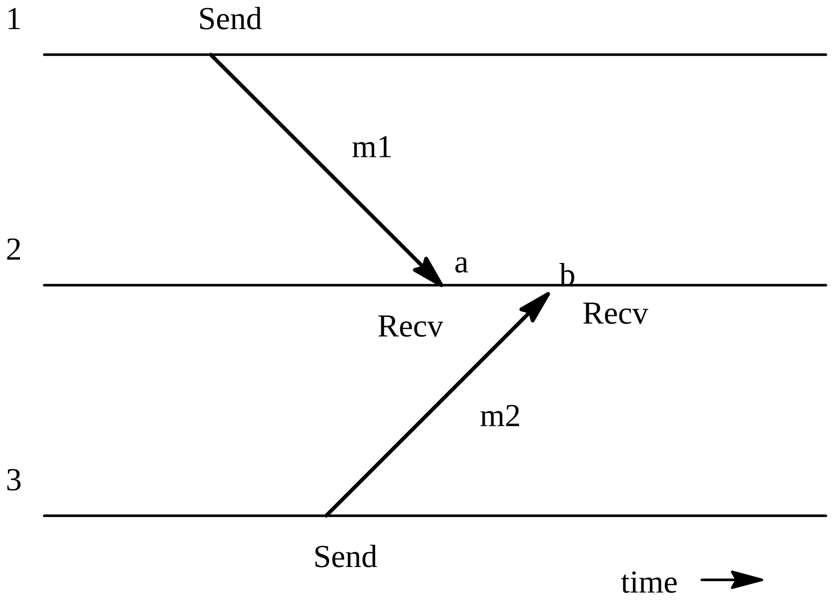


Figure 1. Non–Determinism in Distributed System

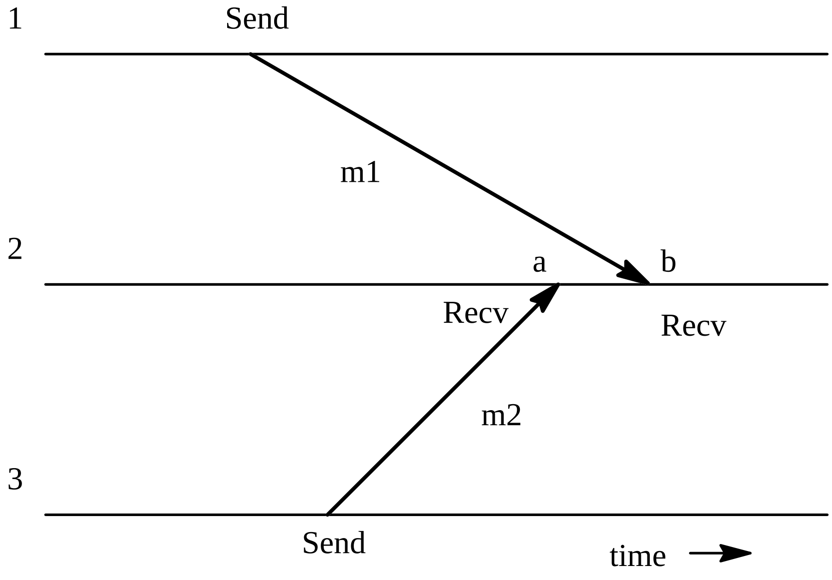


Figure 2. Non–Determinism in Distributed System

To facilitate cyclic debugging, the event histories in distributed programs are recorded. Events usually are the Send and Receive events. The event histories can then be used for re–executing the programs, with the same execution sequence as in the original execution. The event histories eliminate all the non–determinism. Re–executing the distributed programs under the control of event histories is called ‘replay’. The event histories allow the debugger to re–execute the programs such that the order of events is same as that in the original execution. The re–execution can be done in debug mode and more information can be gathered. Additional debug statements can be added and the re–execution will still give the same results. For example, in figure 1, m1 is received by process 2 before m2. This is recorded in the event history. So, no matter what the scheduling delay, network traffic or message delays, m1 will be delivered before m2 during the replay. If message m2 physically arrives at processes 2 before m1, then it is held in a buffer and actually delivered after m1.

This added synchronization can dramatically slow down the programs. In fact some long running programs that send a lot of messages may make cyclic debugging impossible.. For replaying distributed message passing programs, the common strategy is to trace all the messages between processes; so that the execution can be made repeatable. The critical cost in tracing and replaying programs is the cost of tracing messages. In a typical trace and replay scheme, the order in which messages are delivered is first traced during execution. These traces are then used during replay to force each message to be delivered to the same operation as during the traced execution[11].

The algorithms presented in this paper reduce the cost of message tracing, by reducing the messages traced. The basic idea was first proposed by Netzer and Miller [10]. However, their algorithm produces optimal message trace only when the message races are transitive. We improve on this by making the best tracing decisions, given the fact that messages are traced when they are received. In that sense, our message tracing algorithm is optimal. We also improve on the storage requirement for each process, in spite of the fact that, the message tracing decision is made by looking at the complete past history of a process. We also show a simple proof that, no algorithm can exist which will give an optimal trace in general, if messages are traced by looking only at the past history. To our knowledge these results have not been presented before.

2. System Model

The system consists of multiple processes that communicate only through messages. Each process in the system has a unique id which is known to all other processes in the system. The only synchronization events are **Send** and **Recv**. A Send operation can send messages to other process(es), e.g, a unicast or a broadcast. A Recv operation can receive a single message from another process. The Send event specifies the process id to which the message is to be sent in the case of a unicast, or the list of process ids in case of a multicast. The delay in delivering messages is not known. For each process i that can send messages to process j, there is a one–way FIFO channel c, from process i to process j. A Recv event, can receive messages only from the channels that are specified in the event. e.g.; Recv(all) can receive messages over any channel incident on and directed towards the process executing the Recv event, Recv(j,k) can receive messages only from processes j and k. All the channels in the system are first–in–first–out (FIFO). If two messages m1 and m2 are sent by process i to process j, and m1 was sent before m2 then m1 will be received before m2 at process j, although the delay between them is non–deterministic.

The events in the distributed system follow Lamport's [1] 'happened–before' relationships. This relationship denoted by '$\rightarrow$', is an irreflexive transitive closure. The definition of a race between two messages is the same as in [10]. Informally, two messages race if either could have been accepted first by some receive event, due to variations in message latencies or process scheduling. More formally, a message from send event a to receive event b races with message from send event c to receive event d, if and only if there is a frontier that can be drawn, that leads to a frontier race. For details on frontier and frontier races refer [10].

3. Motivation and Related Work

There is plenty of work done in the area of distributed debugging and replaying distributed programs. For replaying distributed message passing programs, the common strategy is to trace all the messages between processes so that the execution can be made repeatable. LeBlanc and Mellor–Crummey [7] suggest a method for distributed debugging called 'Instant Replay', which differs from the strategy of trace and replay. In this method, during program execution, the relative order of significant events is saved as they occur, and not the data associated with such events. As a result, this requires less time and space. The assumption made here is that all the processes are piece–wise deterministic. When the relative order of different IPC events or access of shared objects is saved, then the same data is generated, during replay. It is then guaranteed to reproduce the program behavior during the debugging cycle by using the same input from the external environment and by imposing the same relative order on events during replay that occurred during the original program execution. This technique does not depend on any form of interprocess communication. No centralized bottlenecks are introduced, nor does it require a synchronized global clock. But, a single process cannot be replayed in isolation, all the processes have to be run, as the actual data is not saved.

Netzer and Miller [10] present a technique for tracing and replaying message passing distributed programs, that has a good performance, but is optimal (least number of messages traced) only when the message races are transitive. Their algorithm reduces the messages traced based on the facts that, only messages that race have to be saved and if two messages race, tracing only one of them is sufficient. Run–time tracing decisions are made to trace only a fraction of the total number of messages. This decreases the execution time overhead, as well as the space requirements.

The critical cost in tracing and replaying programs is the cost of tracing messages. In a typical trace and replay scheme, the order in which messages are delivered (but not their contents) is first traced during execution[11]. These traces are then used during replay to force each message to be delivered to the same event as during the traced execution. In the technique presented in [10], a check is made for each message to determine if it races with another message, and only one of the racing messages is traced. When a message is received a race check is performed by analyzing the execution order between the previous receive operation in the same process and the message sender. The ordering information necessary for this check is maintained during execution by appending vector time–stamps onto user messages.

Given that tracing decision is made when the message arrives, the algorithm in [10] does not result in optimal trace, for the general case. The algorithm results in optimal tracing of messages only if all the races are transitive. We call this algorithm as N&M algorithm. It is reproduced here in figure 3.

```
1. Send = event that sent Msg.
2. PrevRecv = previous event (in the same process) willing to receive
from the channel over which Msg was sent.
3. if (PrevRecv [not → ] Send)
trace the message delivered from Send to Recv.
```

Figure 3. N&M Algorithm

Consider the example in figure 1. Messages m1 and m2 race with each other. Now if it is recorded that message m1 was delivered to process 2 during the first Recv and message m2 was delivered during the second Recv; this information is sufficient to replay the processes 1, 2 and 3. However, it is not necessary to record both the messages. If m1 is recorded as the message that was delivered during the first Recv, then m2 has nowhere to go but to the second Recv. It is easily seen that it is optimal (minimal number of messages traced) to trace only one message in this example. It is proved in [10], that only racing messages need to be traced, and we need trace only one message in each race. Non–racing messages cannot introduce non–determinacy and thus their deliveries need not be enforced during replay.

Though this [10] technique provides optimal message tracing in most cases, it is not optimal in all cases. Consider the example in figure 4. The N&M algorithm will trace messages m2 and m3. Whereas, the optimal tracing is when only m2 is traced, as m1 and m3 do not race with each other. This non–optimality manifests itself because in step 3 of N&M algorithm, a blind check is performed irrespective of whether the message received at PrevRecv was traced or not.

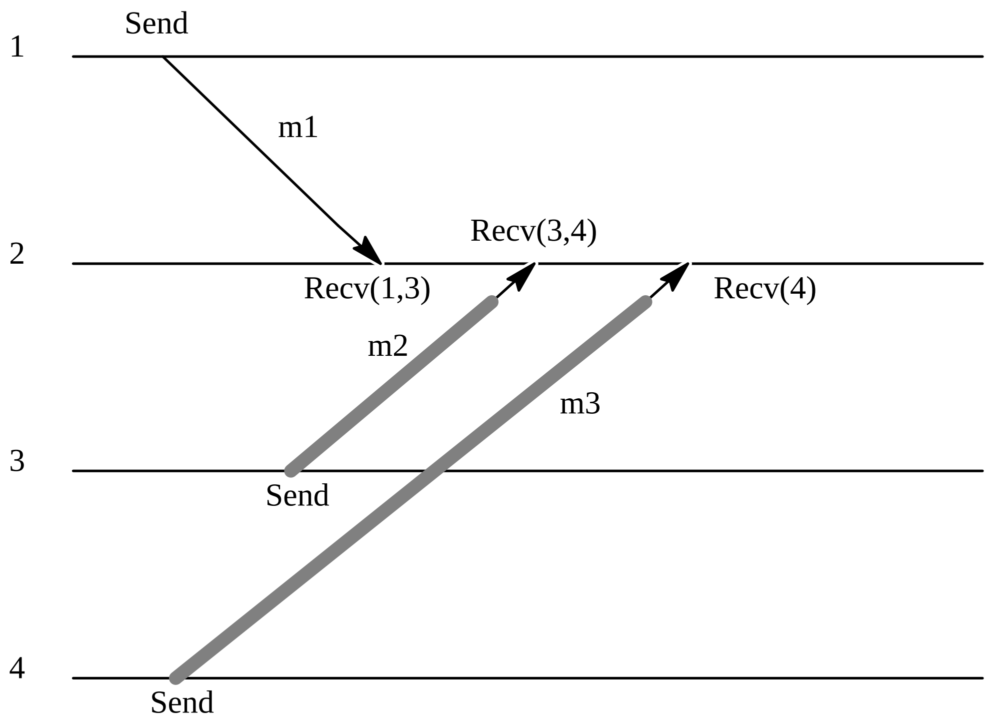


Figure 4. Non–Optimality of N&M Algorithm

In [10] it is shown that the minimum vertex cover problem can be reduced to the problem of finding the optimal message trace. Here each vertex represents a message, and an edge represents a race between two messages. e.g., an edge between vertices A and B means the message represented by A races with the message represented by B. The minimum vertex cover problem is know to be NP–complete, and therefore so is the problem of finding an optimal message trace.

4. Impossibility of Obtaining Optimal Trace with Run–Time Tracing Decisions

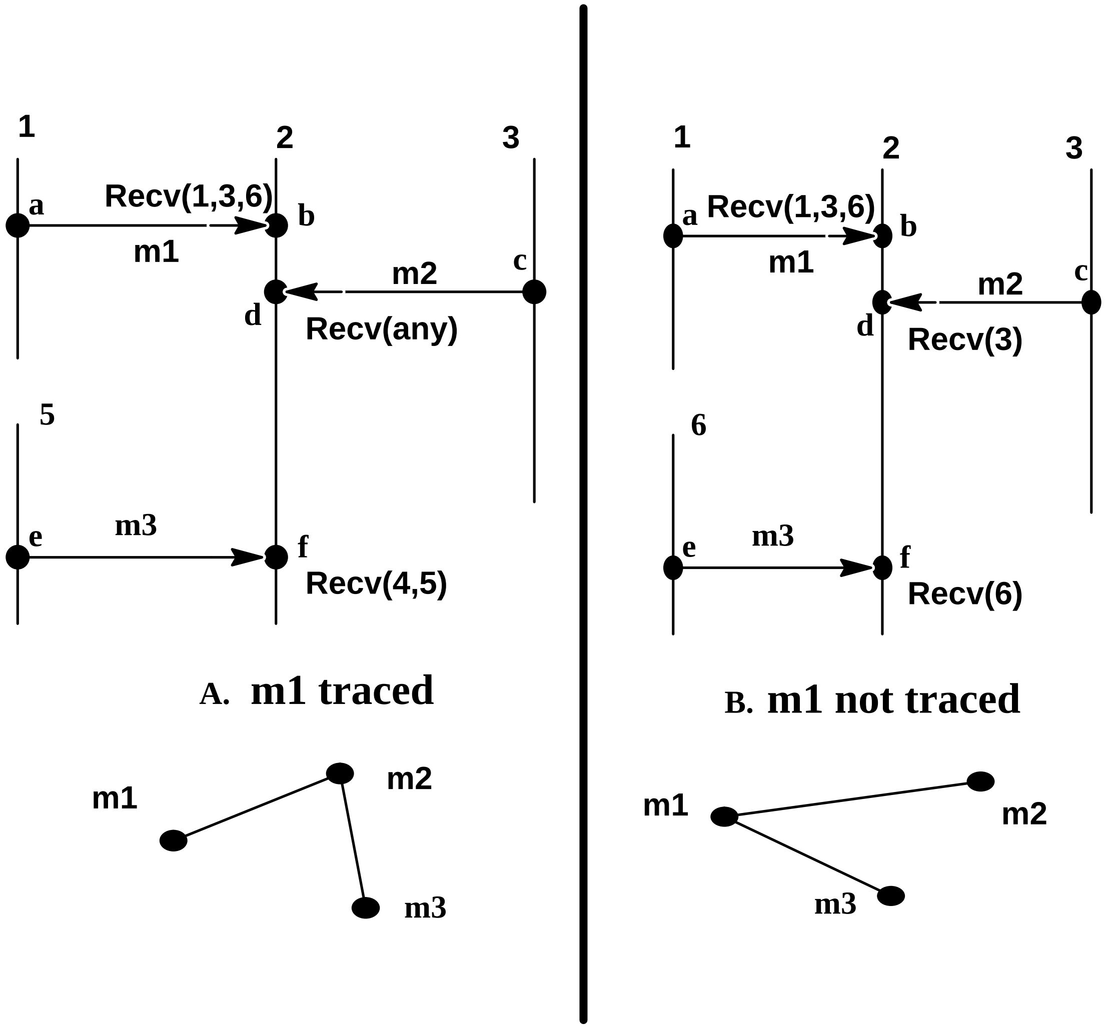
It is impossible to come up with an algorithm that will give an optimal trace of any execution, under the constraint that tracing decisions are made at the instant messages are received, i.e., when a message is traced, no knowledge about the future is available. Also, this means that if a message is not traced when it is delivered, it will never be traced. A decision is made at run–time whether a message is traced or not. Once a decision is made it cannot be changed in the future.

A simple proof (by contradiction) to support this impossibility claim is given next. Let us assume that such an algorithm exists and gives an optimal trace, let us call it Opt_Alg. It is sufficient to give an example that contradicts this assumption. Consider the example given in figure 5. Figure 5.A. shows the execution of process 2. At event b, message m1 is received. Now Opt_Alg will make a tracing decision at event b without the knowledge of the future. The decision has a binary value, either to trace or not to trace.

Case 1: Opt_Alg traces message m1. Now let the future after event b unfold as shown in figure 5.A. As shown in figure 5A, the minimum vertex cover is just vertex m2. Opt_Alg has traced m1 already, so to remove the non–determinacy, it will have to either trace m2 or m3. Either choice results in a non–optimal trace leading to a contradiction.

Case2: Opt_Alg does not trace m1. Now let the future after event b unfold as shown in figure 5.B. As shown in figure 5.B., the minimum vertex cover is just vertex m1. Opt_Alg has not traced m1, so to remove the non–determinacy, it will have to trace m2 and m3 resulting in a non–optimal trace, leading to a contradiction.

Thus, we conclude that Opt_Alg cannot exist.



A. **m1 traced**

B. **m1 not traced**

Figure 5. Impossibility of Obtaining Optimal Trace, with Run–Time Decision.

5. Algorithms for Run–Time Tracing

The basic ideas behind our algorithm are: 1) A message has to be traced only if it is involved in a race with another message. 2) If a message is traced, then it should not be considered for future races.

The algorithm is based on the fact that, if two messages race with each other then only one of them needs to be traced, for replay of programs [10]. If a message does not race with any other message then it need not be traced. If two messages m1 and m2 (refer figure 1) race with each other then for repeatable execution of the program it is sufficient to trace just one of them [10]. If m1 is traced, then during replay, if m2 arrives before m1 (because of scheduling and message delays) the receive event *a* will not accept m2. It will wait till m1 arrives, accept m1 and then m2 will be accepted at event *b*.

If m2 is traced, then during replay, if m2 arrives before m1 (because of scheduling and message delays) m2 again will not be accepted at event *a*. The receive event will wait till another untraced message (m1) arrives.

The above can be implemented as follows: when a message is traced its Send Sequence Number (SSN) and Receive Sequence Number (RSN) are recorded. During replay, if a message was traced then the corresponding send event is modified, so that the message sent during replay is tagged with its SSN and RSN. At the receiver end, the receive event is also modified, so that it receives a message with the same RSN as in the original execution. Now, if the tagged message arrives early, other receive events will not receive the message because the RSN will not match. If the message arrives late, its corresponding receive event will be waiting for this message, and will reject all other messages with incorrect SSN.

We will first present a naive algorithm called Algorithm A, that incorporates the above ideas, and then improve on it in Algorithm B. Algorithm A has excessive storage storage requirement and a long execution time, which are eliminated in Algorithm B.

5.1 Algorithm A

To describe this algorithm, we need to first define some data structure. The system consists of n processes communicating with each other through messages. Each process i, can therefore, receive messages from (n–1) processes over (n–1) channels. Each process i maintains (n–1) linked lists LL_j, $(1 \le j \le n, j \ne i)$ for each channel j, from process j to process i. When a message m1 is received by process i at receive event e1, the receive event is added to some of the linked lists depending on the event e1 that received that message. For all j, (1 ≤ j ≤

n, $j \neq i$), if the receive event e1 could have received a message over channel j, then event e1 is inserted at the head of linked list LL_j. For example, if the receive event e1 was Recv(all), insert e1 in all the linked lists, whereas if the receive event was Recv(j,k,l), then insert e1 in LL_j, LL_k, LL_l. Algorithm A is given in the form of a C–like pseudo–code in figure 6 and explained below.

```
Algorithm_A (NewSend, NewRecv) {
    /* NewMsg is the message sent by event NewSend
     * to event NewRecv
     */
    trace = FALSE
    for j = 1 to n–1
            if (NewMsg could have been received over channel j)
                        trace_decision(j, NewRecv, NewSend)
                        insert NewRecv in $LL_j$
    end for
    if trace = TRUE
            {trace NewMsg}
}

trace_decision(j, NewRecv, NewSend) {
    PrevRecv = head of $LL_j$
    do until tail of $LL_j$ is reached
            if (PrevRecv $\rightarrow$ NewSend)
                        break;
            else if PrevRecv not traced
                        trace = TRUE
                        break;
                    else
                        PrevRecv = next element of $LL_j$
}
```

Figure 6. Algorithm A.

The algorithm is invoked by any process i, whenever a message **NewMsg** sent by event **NewSend** at process k ($k \neq i$) is received by event **NewRecv** at process i. The variable **trace** is used to store the tracing decision which has a binary value. For each channel j over which **NewMsg** could have been received at event **NewRecv**, the function **trace_decision( )** is called and the **NewRecv** event is inserted in the corresponding LL_j. In the function

trace_decision( ), a check is performed for a 'happened–before' relation between events **PrevRecv** and **NewSend**, where **PrevRecv** is the receive event at process i, which 'happened–before' **NewRecv**. This **PrevRecv** event is obtained from the head of the linked–list $LL_{j.}$ If **PrevRecv** 'happened–before' **NewSend**, it implies that the **NewMsg** does not race with any message on channel j and need not be traced. If there is no 'happened–before' relation and if the message received at event **PrevRecv** was not traced, then **NewMsg** is traced. If the message received at **PrevRecv** was traced, then the algorithm does the same check for the previous element in the linked list, till the tail of the list.

5.2 Example 1

Consider the previous example of figure 4, when m1 is received, the corresponding receive event is inserted in LL_1 and LL_3 at process 2. m1 is not traced as there are no **PrevRecv**s. When m2 is received, there is no 'happened–before' relation between the receive at m1 and the send of m2, and since m1 is not traced, m2 is traced. The receive event of m2 is also inserted in LL_3 and LL_4 at process 2 and marked as traced. When m3 is received, it is seen that m3 races with m2, but m2 is already traced. Also, m3 does not race with any other message in the linked list LL_4, so m3 is not traced. The receive event at m3 is inserted in LL_4. At the end of receive of m3 the linked lists at process 2 are as shown below.

LL_1: m1; LL_3: m2–m1; LL_4: m3–m2;

The difference between N&M and this algorithm is that when two messages race, the new algorithm checks whether the previous message was traced. If it was traced then the algorithm goes back in time to see if the current message raced with any other message in the past.

5.3 Example 2

Refer figure 7. When m1 is received, it is not traced, as there are no messages received before it. m2 is traced as it races with m1 and m1 is not traced. When m3 is received, a race check is made with m2. Although m3 races with m2, m2 was traced therefore, the race between m2 and m3 is ignored. So we go back in past and see that m3 raced with m1 and m1 was not traced; resulting in the tracing of message m3. This is again an optimal trace given that tracing decisions are made when the messages are received. The linked lists at the end of the algorithm are as shown below:

LL_1: m3–m2–m1; LL_3: m3–m2–m1; LL_4: m3–m2–m1;

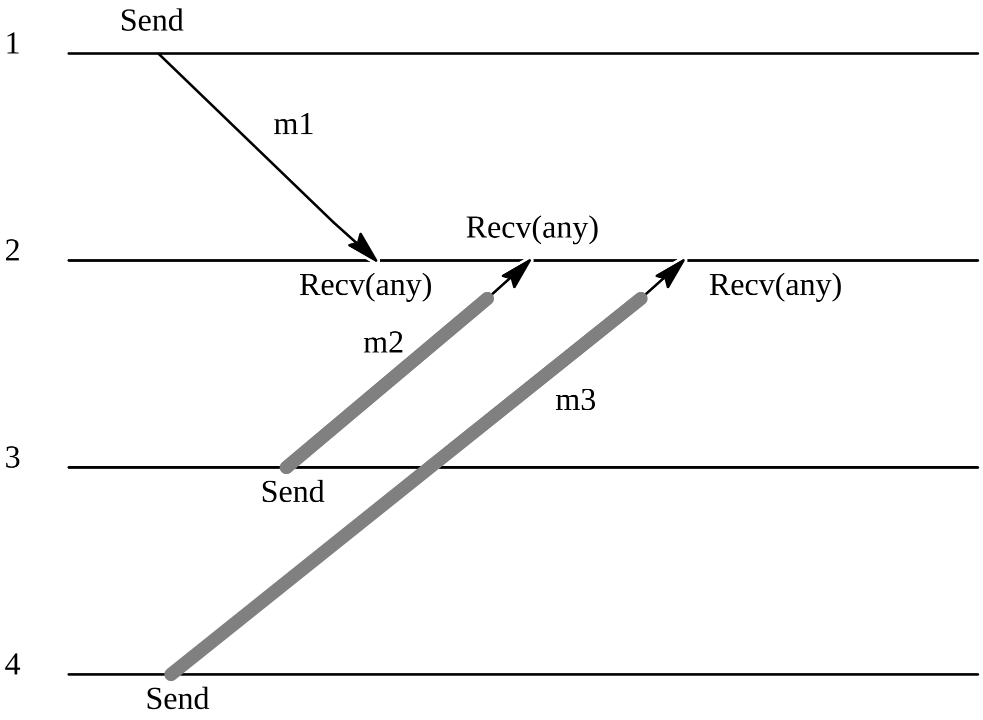


Figure 7. Example 2.

The correctness of the algorithm follows from [10] and the observation that traced messages effectively do not race with any other message.

5.4 Algorithm B

For algorithm A, it can be seen that all the information about the past receive events is saved in the linked lists. This will result in enormous wastage of memory. For long running programs or programs with lot of message passing activity, this may render tracing and replaying impossible. If a condition is found which will allow past receive events to be purged from the lists, without affecting the correctness or optimality of the algorithm, then the number of receive events stored in the lists can be limited. As shown in Theorem 1, we need to keep only one message per channel, per process. I.e., (n–1) linked lists per process can be replaced with just an array of size (n–1) for the past receive events.

Theorem 1: For a message passing system, given that tracing decisions are made at the instant when a message is received at a process, for optimal tracing of messages, a check has to be made only with C number of receive events, in the worst case, where C is the number of channels incident on and directed towards the process.

In other words, only C receive events need to be stored per process, for optimal tracing.

Assumptions:

A1. There are n processes in the message passing system.
A2. Tracing decision is made at each process i, when it receives a message (decision is made with no information about the future).
A3. All channels are FIFO.
A4. A process i can receive messages from (n–1) processes over (n–1) channels. Where $1 \le i \le n$.

Lemma 1: A message from process j to processes i will not race with any other message from process j to processes i. In other words, a message from process j to process i can race with messages from (n–2) processes only.

This result follows directly from [1] and A3. Since all events in a single process are totally ordered, if two messages are sent from process j to process i, there is a 'happened before' relation between the two send events in process j. From assumption A3, the message sent at the first 'Send' event will always be delivered before the message from the second 'Send' event. In figure 8, m1 will always be delivered before m2. Thus, a message from a process races only with messages from (n–2) processes.

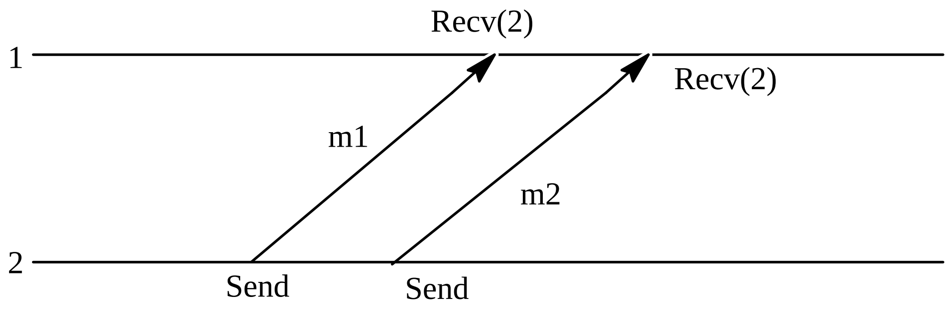


Figure 8. Messages from the same process do not race with each other.

Lemma 2: If a message is traced, effectively, it does not race with any other message.

If a message is traced, then in the replay, the corresponding Receive event will not receive messages from any other Send event but the one from which it received the message in the original execution, which was traced. This means that for the algorithm, a traced message need not be considered for race checks.

Lemma 3: A message from process j to process i can race with a message from process k to process i, only if at least one receive event (out of the two Receive events) was ready to receive a message from either process.

This is based on the functionality of the function 'Recv'. It is obvious from the fact that if a receive event was Recv(j), it will not receive messages from any other channel but that from process j to process i, whereas a message received at the event Recv(j, k, ...) can potentially race with messages from other processes.

Proof of the Theorem:

Let us assume that message m_t from process j to process i races with t messages (m_0, m_1, ..., $m_{(t-1)}$) from process k to process i. (We are considering messages that were received only in the past from assumption A2). All t messages are not traced (if they were, then by Lemma 2 above, they will not race with m_t). By assumption A2, tracing decision is made at the instant the message arrives; this implies that m_0, m_1, ..., $m_{(t-1)}$ will never be traced (as they belong to the past). The decision to be made is: should m_t be traced or not? The decision is 'YES' if m_t raced with any of the $m_0, m_1, ..., m_{(t-1)}$ messages else it is 'NO' (unless it races with a message from some other process). Without loss of generality we can assume that the messages received from process k be in the time order $m_0, m_1, ... ,m_{(t-1)}$. It will never happen that m_t races with any of $\{m_0, m_1, ..., m_{(t-2)}\}$, but not with $m_{(t-1)}$. This can be proved by contradiction.

Let us assume that m_t races with at least one of $\{m_0, m_1, ..., m_{(t-2)}\}$ say m_a, but not with $m_{(t-1)}$ $(a < t-1)$. By this assumption we have Receive of $m_{(t-1)} \rightarrow$ Send of m_t, because $m_{(t-1)}$ and m_t do not race and Receive of m_t could NOT have occurred before that of $m_{(t-1).}$ But, Receive of $m_a \rightarrow$ Receive of $m_{(t-1)}$. By transitivity Receive of $m_a \rightarrow$ Send of $m_{t.}$ Leading to a contradiction that m_t did not race with m_a.

The tracing decision depends only on whether m_t races with $m_{(t-1)}$. i.e., we need to keep only one untraced receive event per channel per process.

For algorithm A, we keep (n–1) linked–lists, one for each channel, in every process. In each list we keep all the Receive events that received a message with a potential race condition over that channel. From the above result, we never need keep more than (n–1) Receive events in each list, one for a message from each process.

In each list there will never be more than (n–1) Receive events as proved above, but the messages corresponding to the Receive event are from different processes and they race with each other. But, if they race with each other, then all of them (but one) would have been traced. Therefore, there will never be more than one Receive event in each list. Thus proving theorem 1. It follows that each process will have maximum of (n–1) Receive events stored from its past.

□

5.5 Example 3

One may tend to think that the above result is incorrect, as it is not very intuitive. Consider the example shown in figure 9. It can be easily seen that if m3 races with m2, it may or may not race with m1. But, if m3 does not race with m2 (there is a ‘happened before’ relation between receive of m2 and that if m3), then m3 does not race with m1 either. Thus it is sufficient to determine if m3 races with m2. We therefore do not need to save the receive event of m1. Thus, a race check with m2 will result in tracing of message m3. If for some reason m2 raced with another message (received before m3, e.g. *m’*) and was traced, then we do not need to save the receive event of m2 as it does not race with any message (by Lemma 2). When m3 is received, a race check is performed with receive event of m1, which will again lead to the tracing of m3.

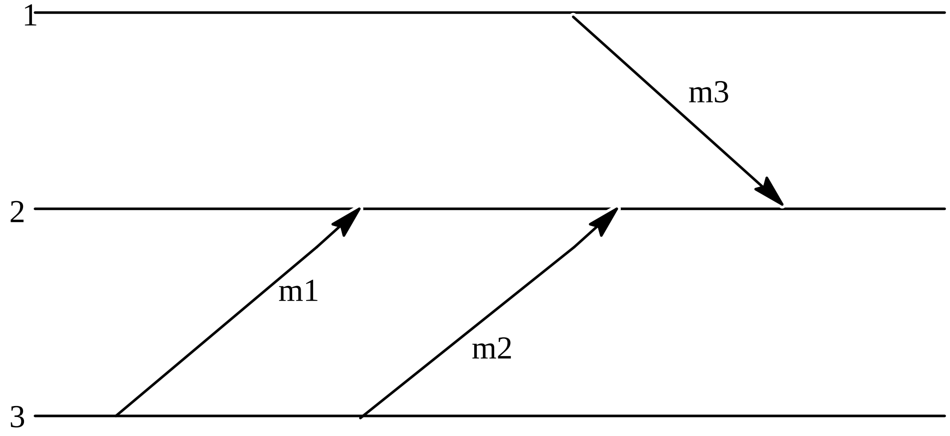


Figure 9. Example 3

Algorithm B is given in figure 10. The algorithm is invoked by any process i, whenever a message **NewMsg** sent by event **NewSend** at process k (k ≠ i) is received by event **NewRecv** at process i. The variable **trace** is used to store the tracing decision which is a binary value. For each channel j over which **NewMsg** could have been received at event **NewRecv**, a check is made for a ‘happened–before’ relation between the the previous receive event (**LL[j]**) and the **NewSend** event. If there is a ‘happened–before’ relation then it implies that the **NewMsg**

```
Algorithm_B (NewSend, NewRecv) {
    /* NewMsg is the message sent by event NewSend to event
     * NewRecv
     */
    trace = FALSE
    for j = 1 to n–1
            if (NewMsg could have been received over channel j)
                        if LL[j] → NewSend
                                    LL[j] = NewRecv
                        else
                                    trace = TRUE
    end for
    if trace = TRUE
            {trace NewMsg}
}
```

Figure 10. Algorithm B.

does not race with any message on the channel j and need not be traced. The variable **LL[j]** is updated to the **NewRecv** event. If there is no 'happened–before' relation, then **NewMsg** is traced and **LL[j]** is left untouched.

5.6 Example 4

In section 4 we showed that its impossible to obtain an optimal trace, given the fact that tracing decision is made at the instant a message arrives at the destination. This example illustrates the non–optimality of algorithm B. In figure 7, if the second Receive event is changed to Recv(1,3) and the third Receive event is changed to Recv(1,4), then the trace obtained by algorithm B is non–optimal. The optimal trace would be to trace just message m1, as m1 races with m2 and m3 but m2 does not race with m3. Algorithm B will not trace m1 as there are no 'PrevRecv's. Instead it will trace m2 and m3 leading to a non–optimal trace.

6. Summary and Future Research Directions

In this paper we have presented an algorithm that traces messages optimally, given the constraint that tracing decisions are made at the instant a message is received by a process. We have also improved on the memory requirements for this algorithm. Using this technique, message traces required for distributed debugging can be significantly reduced. This will lead to less debugging overheads, which include reduced memory, storage, and execution time. For long running distributed programs this is very critical. If the debugging overheads are high, it is not possible to debug the programs using cyclic debugging techniques if every message is traced. We have also shown that if tracing decisions are to be made at run–time (i.e., at the instant a message is received by a process), it is impossible to have an algorithm which will trace messages optimally.

Finding an algorithm that will give an optimal trace under every possibility is equivalent to finding the minimum vertex cover, which is known to be NP–complete. One way to do this can be to save all the message that are received by the process. At the end of the execution run an algorithm (which will run in exponential time) to calculate the optimal trace and trace only those message. This can be done even if the process crashed (because of a bug), as the node it is running on has not failed (as opposed to situation occurring for requirements of fault–tolerance). But, this defeats the purpose of producing optimal message trace. The time required by this optimal algorithm will be very large and we are better off instead, by tracing all the messages.

Future work includes implementation of the algorithm given in this paper to see how it affects the number of messages traced for different classes of application and the amount of actual time saved. This can then be compared with the implementation results in [10]. Better tracing strategies include considering algorithms which make tracing decisions with the knowledge about the events in a future window, where the future window is of fixed (or varying for adaptive algorithms) size. The size of the window will have to be determined from experimental results or some heuristics. For example, when a tracing decision is made about a receive event ‘a’ in process i, n number (where n is the size of the future window) of events in process i that occurred after event ‘a’ have already occurred. In other words, tracing decision for event ‘a’ is made at an event which is separated by n events in the future.

References

[1] Leslie Lamport, ‘‘Time, Clocks, and the Ordering of Events in a Distributed System,” Communication of ACM, vol 21, no 7, July 1978.

[2] Allen D. Malony and Daniel A. Reed, ‘‘Models for Performance Perturbation Analysis,” ACM/ONR Workshop on Parallel and Distributed Debugging, pp 15–25, Santa Cruz, CA, May 1991.

[3] J. P. Black and M. H. Coffin and D. J. Taylor and T. Kunz and T. Basten, ‘‘Linking Specification, Abstraction, and Debugging,” University of Waterloo, Canada, Tech Report TR–94–02, November 1993.

[4] Stuart I. Feldman and Channing B. Brown, ‘‘IGOR: A System for Program Debugging via Reversible Execution,” Proceedings of the SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging, pp112–123, Madison, WI, May 1988.

[5] Perry Emrath and Sanjoy Ghosh and David Padua, ‘‘Detecting Non–determinacy in Parallel Programs,’’ IEEE Software 9,1, pp.69–77, January 1992.

[6] Charles E. McDowell and David P. Helmbold, ‘‘Debugging Concurrent Programs,” ACM Computing Surveys, vol 21, no 4, pp 593–622, December 1989.

[7] Thomas J. LeBlanc and John M. Mellor–Crummey, ‘‘Debugging Parallel Programs with Instant Replay,” IEEE Transactions on Computers, C–36, 4, pp 471–482, April 1987.

[8] Colin Fidge, ‘‘Fundamentals of Distributed System Observation,” The University of Queensland, Australia, Tech Report 93–15, November 1993.

[9] Z Yang and T.A. Marsland, ‘‘Global Snapshots for Distributed Debugging: An Overview,’’ University of Alberta CS, Tech Report TR 92–03, 1992.

[10] Robert H. B. Netzer and Barton P. Miller, ‘‘Optimal Tracing and Replay for Debugging Message–Passing Parallel Programs,’’ Proceedings of Supercomputing ’92, Minneapolis, MN, pp 502–511, November 1992.

[11] Larry D. Wittie, ‘‘Debugging Distributed C Programs by Real Time Replay,’’ Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging, pp57–67, vol 24, no 1, January 1989.

[12] Robert H. B. Netzer, Sairam Subramanian and Jian Xu, ‘‘Critical–Path–Based Message Logging for Incremental Replay of Message–Passing Programs,’’ International Conference on Distributed Computing Systems, 1994.