Partial Temporal Ordering in Distributed Network

Soumen Saha
Haldia Institute of Technology
Haldia, West Bengal

Utpal Roy, Ph.D
Department of Computer & System Sciences
Siksha-Bhavana, Visva-Bharati

ABSTRACT

As communication is important, it is not the entire thing rather something more interesting as well as complicated. Closely related is how processes cooperate and synchronize with one another. In a distributed system an application may have several processes that run concurrently on multiple nodes of the system. For correct results, several such distributed applications require that the clocks of the nodes are synchronized with each other. For concurrency we have used vector clock method .But there are several disadvantages. For that we have developed a new algorithm for vector clock method from which we can define the concurrency among processes. In our proposed algorithm the vector of each process's local clock consists of (n+1) parameter where n is the number of processes in the system. The (n+1) th parameter is used as a flag which help to discuss the concurrency among different processes..

General Terms

Distributed System, Vector Clock, Synchronized Clock, Happened Before, Time-stamp, message Passing.

Keywords

Distributed system, Ordering, Vector Clock, Lamport's Algorithm, PVM(parallel virtual Machine), Linux..

1. INTRODUCTION

A distributed system consists of a collection of autonomous computers, connected through a network and distribution middleware, which enables computers to coordinate their activities and to share the resources of the system, so that users perceive the system as a single, integrated computing facility. The main characteristic of these computations is that the processes do not already share a common global memory and that they communicate only by exchanging messages over a communication network. Moreover, message transfer delays are finite yet unpredictable. This computation model defines what is known as the asynchronous distributed system model, which includes systems that span large geographic areas and are subject to unpredictable loads. A model distributed execution can be exemplified as follows: suppose a distributed program made up of n sequential local program. When these programs are executed they are unable to communicate among themselves and synchronize themselves by exchanging messages among themselves. Executing a local program gives rise to a sequential process. Let l_1, l_2,….., l_n be this finite set of processes. We assume that, at runtime, each ordered pair of communicating processes(l_i,l_j) is connected by a reliable channel c_ij through l_iwhich can send messages to l_j Executing an internal, send, or receive statement produces an internal, send, or receive event.

Let〖 e〗_i^x (x≥1) be the x th event process l_i produces. The sequence constitutes the history of〖 l〗_i . Let S be the set of events that a distributed computation produces. This set is structured as a partial order by famous L. Lamport's "happened-before" relation,[1] denoted " →" and defined as e →f means that event e can affect event f. Consequently, ¬(e → f) means e cannot affect f. The partial order constitutes a formal model of the distributed computation with which it is associated. The states of the events in distributed system are related with vector clock. In a distributed system the vector clock system is a mechanism that associates timestamps with events (local states) such that comparing two events' timestamps indicates whether those events (local states) are causally related (and, if they are, which one comes first). In the time-stamping system, each process l_i has a vector of integers 〖VC〗_i {1….n} (initialized to {0,0,….,0}) that can be maintained in proper manner.

The language of partial order time expresses many issues central to many problems in asynchronous distributed system. Traditionally we regard time as a scalar value, totally ordering on the events in a system. However the very nature of asynchronous distributed systems suggests that we should use an order that is partial, not total so that we can deliberately leave unordered two separate events that have no knowledge of each other. In this partial order time model, both the presence and the absence of a path between two events carry meanings whether one event necessarily precedes the order or they are concurrent. If we use merely a total order, we may lose the latter information.

We know processes in distributed system[2] communicates with one another using several layered protocols, request/ reply message passing (RPC), and group communication. While communication is important, it is not the entire story. Closely related is how processes cooperate to each other and synchronize with one another. In a distributed system an application may have several processes that run concurrently on multiple nodes of the entire system. For correct results, several such distributed applications require that the clocks value of the nodes are synchronized with each other. In a distributed system, synchronized clocks also enable one to measure the duration of distributed activities that start on one node and may be terminate on another node. The present study is devoted towards discussion and development of a new algorithm than can explain concurrency in the field of partial temporal ordered system involving vector clock.

2. 1.1 Total order:

In mathematics[4] a total order or linear order on a set A is any binary relation on A that is antisymmetric, transitive, and total. This means that, if we denote the relation by R, the following statements hold for all a, b and c in A:

if a R b and b R a then a = b (antisymmetry)

if a R b and b R c then a R c (transitivity) a R b or b R a (totalness)

A set with a total order on it is called a totally ordered set, a linearly ordered set, or a chain. The totalness property can be stated thus: that any pair of elements in the chain is mutually comparable.

1.2 Partial order:

A partial order[4] is a binary relation R over a set B which is reflexive, antisymmetric, and transitive, i.e., for all a, b and c in B, we have that:

a R a (reflexivity);

if a R b and b R a then a = b antisymmetry);

if a R b and b R c then a R c (transitivity).

A set with a partial order is called a partially ordered set. The term ordered set is sometimes also used for posets, as long as it is clear from the context that no other kinds of orders are meant

1.3 Ordering on the basis of Logical Clock:

We would like to order the events according in a distributed system in such a way as to reflect their possible connections. Certainly if an event A happens before an event B, then A cannot have caused by B. In this situation we cannot say that A is the directed cause of B, but we cannot exclude that A might have influenced B. We want to characterize this "*happens before relation"* on events described in the landmark paper of L. Lamport [1]. Here only two kinds of events have been considered, the sending of a message and the receiving of a message.

a) If events e_1 and e_2 occur in the same system and e_1 occurs before e_2 (there is no problem to determine this in a single system) then e_1 happened-before e_2, written $e_1 \rightarrow e_2$.
b) If event e_1 is the sending of a message and e_2 is the receiving of that message, then e_2 *happened before* e_2 .
c) If $e_1 \rightarrow e_2$ and $e_2 \rightarrow e_3$ then $e_1 \rightarrow e_3$.

The relation is "→" a partial order. Given events e_1 and e_2 it is not true that either they are the same event or one happened before the other. These events may be unrelated. Events that are not ordered by happened before are said to be concurrent. This characterization of happened before is un satisfactory since, given two events, it is not immediate (think of the time it takes to evaluate a transitive relation) to determine if one event happened before the other or if they are concurrent. We would like a clock C that applied to an event returns a number so that the following holds.

A vector timestamp $V(a)$ assigned to an event 'a' has the property that if $V(a) < V(b)$ for some event 'b' , then event 'a' is known to causality precede event b. Vector time stamps are constructed by letting each process P_i maintain a vector V_i with the following two properties-

1) $V_i(j)$ is the number of events that have occurred so far at P_i.
2) If $V_i(j) = k$ then P_i knows that k events have occurred at P_i.

The rules on vector clocks in a system with n computers:-

1) Each computer starts with a local clock set at $(0,0,0,\dots\dots\dots)$
2) When on computer i there is a sending event, increment the ith component of the clock by 1 leaving other components unchanged, then tag both the events and the message with this value.
3) When on computer i there is a receiving event, form a new local clock value taking the component wise maximum of the local clock and the time stamp on the arriving message. Then increment by 1 the i th component. Finally tag the event with this value.
4) We consider here, clock unit is different for each processor.

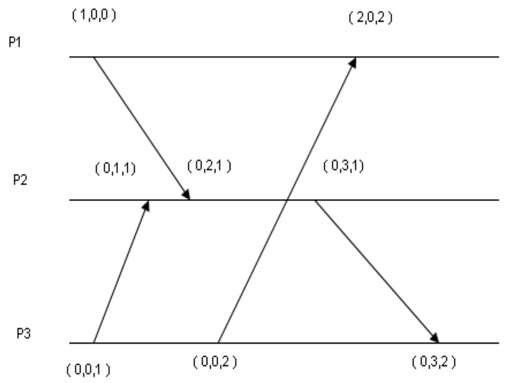


Fig1: A simple vector clock example

2. Our proposed Work

Here a new as well as different algorithm for implementing a vector clock has been proposed. The algorithm is described below.

Algorithm for new vector clock method is-struct x

```
{
    int vector[p];   //  where n is the number of      //processes
and p=n+1
} number[n];


Algorithm New_Vector_Clock( n )
{
    node:=n;
    for I:=1 to n   do
      {
         for j:=1 to (n+1)    do
                 number[i].vector[j]:=0;
      }
while (there is a sending event on i th computer)   do
{
   number[i].vector[i]:=number[i].vector[i]+1;
   number[i].vector[n+1]:=1;     // Flag is set
   Tag both  event and message with this value;
 }
while (there is a receiving  event on i th computer )    do
{
      Take the component wise maximum of the  local clock;
      number[i].vector[i]:=number[i].vector[i]+1;
      Tag both  event and message with this value;
 }
 for i:=1 to n   do
   for j:=1 to  n     do
     {
      if    (    number[i].vector[n+1]==1    &&
number[j].vector[n+1]==1 )  then
        Return Process i and Process j as concurrent;
     }
}        // End of  new vector clock algorithm
```

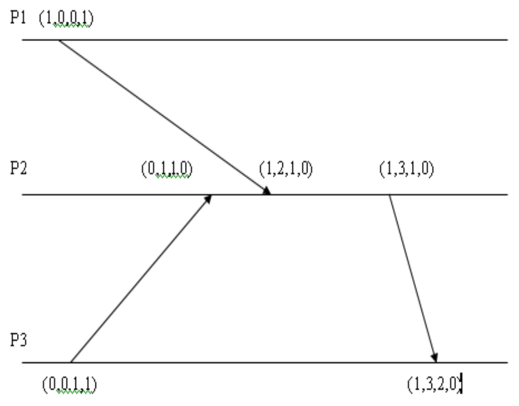


Fig2- Example for proposed vector clock method

3. Complexity Analysis

Let n be the number of processes. Each process has its own local vector clock. The number of parameter in each vector is (n+1). Let m be the total number of events occurred in the system. Whenever there is a sending event on i th computer the i th component and the flag i.e, (n+1)th component of vector are updated. So total number of component updated is 2. On the other hand whenever there is a receiving event on i th computer only the i th component is updated. Hence total number of component updating is 1. Hence we can say that the maximum number of updating for message sending or message receiving event is 2.

Whenever the component updating is done, any 2 arbitrary components from (n+1) component can be updated. For each event total number of updating is   (n+1)C2 .Now there are m maximum events that are maximum m times the updating can be done.

So for m events the total number of updating is

=m *^((n+1)) C_2
=m*(n(n+1))/2
=(m*n^2+m*n)*0.5

Hence the worst case complexity is O(mn^2)

4. Testing

The proposed algorithm has been executed and tested in the PVM (Parallel Virtual Machine[5]). PVM(Parallel Virtual Machine) is compatible in LINUX environment.

PVM is an integrated set of software tools and libraries that emulates a general-purpose, flexible, heterogeneous concurrent computing network on interconnected computers of varied architecture. The overall objective of the PVM system is to enable such a collection of computers to be used cooperatively for concurrent or parallel computation.

The PVM simulating software   is based on the notion that an application consists of several tasks. Each task is responsible for a parts of the application's computational workload. Sometimes an application is parallelized within its functions; that is, each task performs a different function, for example, input, problem, setup, solution, output, and display. This process is often called functional parallelism. A more common method of parallelizing an application is called data parallelism.

The PVM system is consists of two parts. The first part is a daemon, named pvmd3 and sometimes calls pvmd that resides on all the computers making up the virtual machine. pvmd3 is designed so any user with a valid login can install this daemon on a machine. When a user wishes to run a PVM application, he first creates a virtual machine by starting PVM. The PVM application can then be started from a UNIX prompt on any of the hosts. Multiple users can configure overlapping virtual machines, and each user can execute several PVM applications simultaneously. The second part of the system is PVM interface routines. It contains a functionally complete repertoire of primitives that are needed for cooperation between tasks of an application. This library contains user-callable routines for message passing, spawning processes, coordinating tasks, and modifying the virtual machine.

PVM applications can be written in using C , Java and other languages. To program in C using PVM, the we add PVM function calls to the code. The compiled code is then linked with libraries which handle the PVM calls. PVM provides a very flexible environment for message passing. It supports MIMD (Multiple Instruction stream over Multiple Data stream) style parallel computation, though most programs are written in the SIMD (Single Instruction over Multiple Data stream) style.[5]

Simple C Code myprog.c

```
#include "pvm3.h"
#define TASKS 5
main()
{
 int id, i;     /* enroll in PVM sw*/
 id = pvm_mytid();
      /* Possibly do some work here */
```

```
  printf("Hi from task  no%d", id);
        /* exit from PVM sw*/
  i = pvm_exit();
  exit();
}
```

Compiling in C

Compile the code using the GNU C compiler, letting it know the location of the include files, and linking with the PVM libraries:

```
cc -o prog prog.c -PVM_ROOT/include -L$PVM_ROOT/lib/LINUX -lpvm3 –lnsl
```

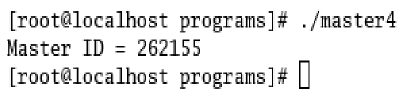
```
pvm> spawn -3 -> prog
[1]
3 successful
t40004
t40005
t40006
[1:t40005] Hi from task no262149
[1:t40005] EOF
[1:t40006] Hi from task no262150
[1:t40006] EOF
[1:t40004] Hi from task no262148
[1:t40004] EOF
[1] finished
```

All PVM tasks are identified by an integer task identifier (TID). Messages are sent to and received from ids. Since ids must be unique across the entire virtual machine, they are supplied by the local pvmd and are not user chosen. Although PVM encodes information into each TID the user is expected to treat the ids as opaque integer identifiers. PVM contains several routines that return TID values so that the user application can identify other tasks in the system. There are applications where it is natural to think of a group of tasks. And there are cases where a user would like to identify his tasks by the numbers 0-(q-1)where q is the number of tasks..

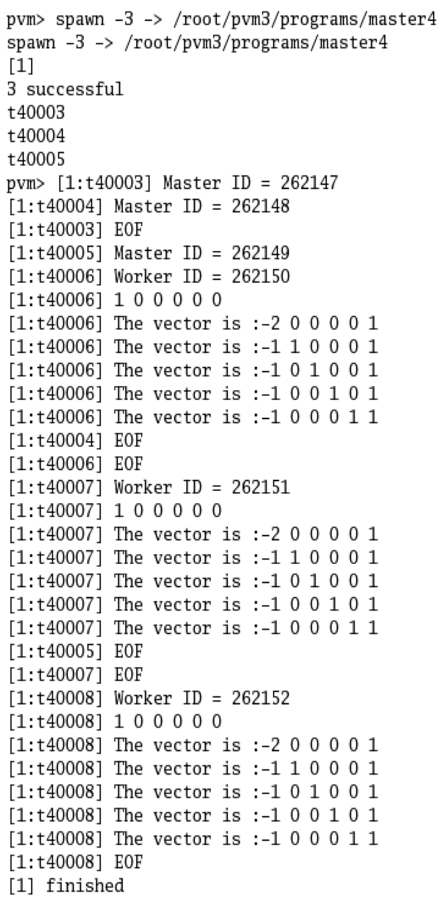
5. Result

For the purpose of simulation we have used the PVM and it yields reasonably justified results. The simulation result is obtained as follows.

The master program is compiled first then the worker program is compiled. The master program is spawned 3 times and the result is shown below. Further, the master program can be run independently. The result is shown below and the elastration of result also is shown at figure 3



```
[root@localhost programs]# ./master4
Master ID = 262155
[root@localhost programs]# 
```



```
pvm> spawn -3 -> /root/pvm3/programs/master4
spawn -3 -> /root/pvm3/programs/master4
[1]
3 successful
t40003
t40004
t40005
pvm> [1:t40003] Master ID = 262147
[1:t40004] Master ID = 262148
[1:t40003] EOF
[1:t40005] Master ID = 262149
[1:t40006] Worker ID = 262150
[1:t40006] 1 0 0 0 0 0
[1:t40006] The vector is :-2 0 0 0 0 1
[1:t40006] The vector is :-1 1 0 0 0 1
[1:t40006] The vector is :-1 0 1 0 0 1
[1:t40006] The vector is :-1 0 0 1 0 1
[1:t40006] The vector is :-1 0 0 0 1 1
[1:t40004] EOF
[1:t40006] EOF
[1:t40007] Worker ID = 262151
[1:t40007] 1 0 0 0 0 0
[1:t40007] The vector is :-2 0 0 0 0 1
[1:t40007] The vector is :-1 1 0 0 0 1
[1:t40007] The vector is :-1 0 1 0 0 1
[1:t40007] The vector is :-1 0 0 1 0 1
[1:t40007] The vector is :-1 0 0 0 1 1
[1:t40005] EOF
[1:t40007] EOF
[1:t40008] Worker ID = 262152
[1:t40008] 1 0 0 0 0 0
[1:t40008] The vector is :-2 0 0 0 0 1
[1:t40008] The vector is :-1 1 0 0 0 1
[1:t40008] The vector is :-1 0 1 0 0 1
[1:t40008] The vector is :-1 0 0 1 0 1
[1:t40008] The vector is :-1 0 0 0 1 1
[1:t40008] EOF
[1] finished
```

Fig3: Program out put snap shot

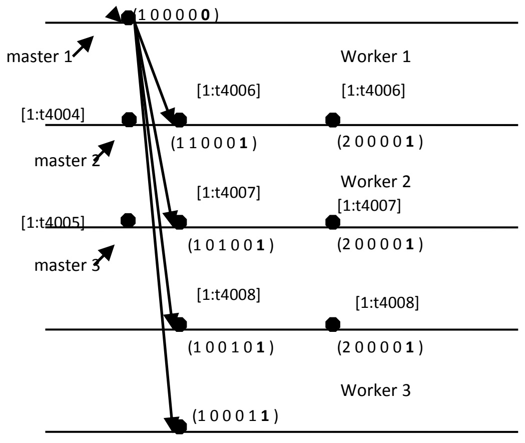


Fig4- Elastration of program output

6. CONCLUSION

The main drawback of a vector clock system is its inability to face the scalability problems. To capture the causality relation among the events that a distributed computation's processes produce, a vector clock system requires vectors of size $n, where\ n$ is the number of processes in the system. To overcome the problem the concept of bounded vector clock has been proposed[6-8]. Further the idea of approximate vector clock is also available in literature.

Here the study has been concentrated on the drawback of the conventional vector clock method that cannot detect the concurrency among different processes.

The present article is a modest approach towards the proposal of a different vector clock which is able to show the concurrency among processes. Here a unique vector clock approach have been proposed which indicate a $flag$ that signifies the concurrency among processes. Further the proposed approach is system independent and can reasonably shows the concurrency among different processes in the system at definite time. In the result it has been shown that, if in a system when more than one process is present then it can easily be detected that which processes are concurrent to which processes.

Hence this approach can be applied to any distributed system, such as banking, cloud computing, reservation system, Parallel processing system etc

7. REFERENCES

[1] L. Lamport, "Time, Clocks and the ordering of Events in a Distributed System," Comm. ACM. Vol 21, No.7 , July 1978, pp 558-565.

[2] A. S. Tanenbaum and M. V. Steen Distributed System : Principles and Paradigms (2nd Edition)

[3] V. Kanakaris, D. Ndzi and D. Azzi, Ad-hoc Networks Energy Consumption: A review of the Ad-Hoc Routing Protocols Journal of Engineering Science and Technology Review 3 (1) (2010) 162-167

[4] S W Smith and J D Tygar, "Signed Vector Timestamps A Secure Protocol for Partial Order Time", Carnegie Mellon Computer Science Technical Report CMU-CS-93-116(1993) .

[5] PVM (Parallel Virtual Machine)A Users' Guide and Tutorial for Networked Parallel Computing; http://www.netlib.org/pvm3/book/pvm-book.html.

[6] L Lamport, P M Melliar-Smith , Byzantine Clock Synchronization: Proceedings of the third annual ACM symposium on Principles of distributed computing PODC 84 (1984).

[7] F.J. Torres-Rojas and M. Ahamad, " Plausible Clocks: Constant Size Logical Clocks for Distributed System," Proc. 10th Int'l Workshop Distributed Algorithms, Springer Verlag, New York, 1996, pp71-88.

[8] R. Baldoni and G. Melideo, Tradeoffs in Message overhead versus Detection Tim in Causality Tracking, tech, report 06-01, Dipartimento di Informaticae Sistemistica, Univ. of Rome, 2000.