ResearchGate

See discussions, stats, and author profiles for this publication at: https://www.researchgate.net/publication/221134159

An Efficient Implementation of Vector Clocks in Dynamic Systems.

Conference Paper · January 2006

Source: DBLP

CITATIONS
10

READS
2,523

4 authors, including:

Xinli Wang
Michigan Technological University
26 PUBLICATIONS **231** CITATIONS
SEE PROFILE

Jean Mayo
Michigan Technological University
58 PUBLICATIONS **437** CITATIONS
SEE PROFILE

Wei Gao
663 PUBLICATIONS **12,829** CITATIONS
SEE PROFILE

Some of the authors of this publication are also working on these related projects:

ThreadMentor View project

leaf pigment estimation View project

All content following this page was uploaded by Xinli Wang on 05 June 2014.

The user has requested enhancement of the downloaded file.

An Efficient Implementation of Vector Clocks in Dynamic Systems

Xinli Wang[a] Jean Mayo[b] Wei Gao[a] James Slusser[a]

[a] USDA UV-B Monitoring and Research Program, Natural Resource Ecology Laboratory
Colorado State University, Fort Collins, CO, 80523-1499

[b] Department of Computer Science, Michigan Technological University, Houghton, MI 49931 USA

Abstract

A system of vector clocks is strongly consistent and it captures the happened before relations among events in the system. These clocks underlie solutions to a number of problems in distributed systems including, among others, detecting global predicates, debugging distributed programs, causally ordering multicast messages, and implementing a distributed shared memory. In general, a data structure of size n, where n is the number of processes in the system, has to be maintained at each process and attached with each message communicated in the system to implement vector clocks. This is a considerable communication overhead in large systems. A differential technique has been proposed to reduce this required communication overhead for static systems with FIFO channels.

In this study, the differential technique is improved to further reduce the required communication overhead. A protocol is proposed to maintain a virtual network topology of a logical ring combined with multiple computation trees so that the differential technique can be applied to dynamic systems. When a process leaves the clock maintained at this process is taken over by another one in the system. At the time a process joins the system, it will inherits the causality relations maintained at the process that creates the new process. Correctness of the protocol and the clock properties are proved as well.

Key words: vector clock, differential technique, dynamic system

1. Introduction

Ordering the events occurring in a distributed computation is fundamental to reasoning, analyzing, and drawing inferences about the computation [5, 9, 12]. Fidge [3, 4] and Mattern [10] independently proposed *vector clocks* to capture Lamport's *happened before* relation [9], which expresses the ordering imposed by the sequential execution of events at each process and the message passing that takes place among processes and is commonly used to order these events. Although this mechanism has a limitation when vector timestamps are used to reconstruct a distributed computation where message overtaking may occur [6], it is strongly consistent [12] and provides a way to precisely capture the whole causality relationships between events occurring in a distributed computation [1, 4, 5, 10, 12]. Vector clocks have been applied to many problems in distributed systems, such as detecting global properties, debugging distributed programs, ordering multicast messages, and implementing a distributed shared memory.

One drawback in the implementation of vector clocks is the required communication overhead. When a message is transferred an overhead of size n is added. For a big system this overhead is considerable, especially when processes can be created and may terminate dynamically because of the unlimited increase in

Citation: Proceedings of The 2006 International Conference on Parallel & Distributed Processing Techniques and Applications & Conference on Real-Time Computing Systems & Applications (PDPTA'06), Vol II, Eds: Hamid R. Arabnia, Maria Artishchev-Zapolotsky, Steve C. Chiu, Yefim Dinitz, Kazuki Joe, Mario Nakamori, Las Vegas, Nevada, USA, June 26-29, 2006, CSREA Press

the vector size with process creation. Although some alternatives are available for vector clocks under certain constraints [7, 11, 14, 16], a data structure of size n is necessary to capture the causal relationships between the events [2] in asynchronous systems.

One approach to reduce the communication overhead is the "differential technique" which was discussed by Fidge [4] and developed by Singhal and Kshemkalyani [12, 15] for systems with FIFO channels. Under this technique, when process p_i sends a message to p_j only those components of the vector clock at p_i that have changed since last time p_i sent a message to p_j are piggybacked with this message. Hélary *et. al* [8] extended Singhal and Kshemkalyani's protocol for the systems without FIFO channels.

The differential technique can be improved by observing that some of the changed elements were modified because of a message receipt from p_j and it is therefore not necessary to transfer such changed elements to p_j when a message is sent to p_j.

Fidge [4, 5] and Richard [13] independently developed schemes to efficiently implement vector clocks in dynamic systems. However their schemes are good only for special purposes. If multiple processes are leaving the system concurrently, some of the vector clocks maintained at the leaving processes may be lost permanently in both Fidge's and Richard's schemes.

In this study we extend the differential technique to implement vector clocks in dynamic systems. This implementation improves the Singhal and Kshemkalyani's technique so that the communication overhead is further suppressed. A protocol for process creation and termination is proposed and integrated into the implementation so that the vector clock maintained at a leaving/terminating process will not be lost in the case when multiple processes leave the system or terminate concurrently.

2. System Model

A distributed system is modeled with a finite set of processes running on geographically separated machines that are connected with a communication network. The processes cooperate and coordinate through message passing. All processes are not faulty. We assume reliable asynchronous communications over the network. Messages are reliably delivered to their correct destination processes in the order when they were sent. Message delay is finite but unpredictable.

A distributed computation in such a system starts with a nonempty set of processes, which are called *initial processes*. We assume that the initial processes are connected with a logical ring and each of them knows its neighbors. If the edges (p_i, p_j) and (p_j, p_k) exist on the logical ring, then p_i is called the up stream neighbor and p_k the down stream neighbor of p_j. In the progresses of the computation, new processes can be created, external processes can join, and existing processes can leave the system at any time. We assume that at least one initial process exists until the computation terminates. As the computation progresses, a dynamic network topology of multiple trees will be maintained in the system. Each of the trees is rooted at an initial process. A protocol is superimposed upon the computation to implement vector clocks. Let $SYS(t) = \{p_1, p_2, \cdots, p_i, \cdots, p_n\}$ be the process set in the system at real time t, where n is the number of processes.

3. Implementation of Vector Clocks Using Improved Differential Technique

As Hélary *et. al* [8] suggest, we assume that all of the events executed in the system are relative events in implementing vector clocks.

3.1. Data Structures

The following variables are defined at an arbitrary process p_i.

$Parent_i$: the parent of p_i. If p_i is an initial process, $Parent_i$ holds the ID of p_i's up stream neighbor. Otherwise, $Parent_i$ holds the ID of the process who created p_i or accepted p_i while p_i was joining the system.

$Child_i$: the children of p_i. If p_i is not an initial process, $Child_i$ is a set of IDs of the processes that were created or accepted by p_i when they were joining the system and its initial value is an empty set. For an initial process p_i, its down stream neighbor is also included in $Child_i$ as its initial value.

$Leaving_i$: a Boolean variable. When p_i is terminating, $Leaving_i$ is set to $true$; otherwise $Leaving_i$ is set to $false$.

VC_i (Vector Clock): vector clock of p_i. It is a set containing a pair of (j, c_j) for a process p_j in the system that has communicated with p_i. The integer c_j is the scalar clock at p_j in the p_i's point of view. For convenience we use $VC_i[j]$ to denote the value of c_j and $VC_i(e)$ the vector clock VC_i right before event e occurs. Initially $VC_i = \{(i, 0)\}$.

LU_i (Last Updated): a set of the last updated clocks. It contains a triple of (j, k, u_j) for a process p_j in the system with u_j equal to the value of $VC_i[i]$ when p_i last updated $VC_i[j]$. The integer k identifies the process to which the last update of $VC_i[j]$ was related. If this modification was done because of an internal event or a message sending event at p_i, then $k = i$. If this modification was made because a message receipt from p_s, then $k = s$. We use $LU_i[j][0]$ to denote the value of k and $LU_i[j][1]$ the value of u_j. Initially $LU_i = \{(i, 0, 0)\}$.

LS_i (Last Sent): a set of the last sent clocks. It contains a pair of (j, s_j) for a process p_j in the system with s_j equal to the value of $VC_i[i]$ when p_i last sent a message to p_j. We use $LS_i[j]$ to denote the value of s_j. Initially $LS_i = \{(i, 0)\}$.

TVC_i (Terminated Vector Clocks): vector clocks of terminated processes. It is a set of vector clocks that were maintained at terminated processes. Initially $TVC_i = \emptyset$.

3.2. Protocol for Updating the Data Structures

In this study, messages fall into two categories: (1) computation messages that are related to the distributed computation; (2) termination and creation notification messages that are transmitted while a process is terminating or being created. We consider only computation messages in this subsection and the latter will be discussed in the next subsection.

In the following exposition, a differential vector clock of p_i relative to p_j is defined as a such vector that contains a pair of (k, d_k) for each process p_k that the value of $VC_i[k]$ has been updated since last time p_i sent a message to p_j and this modification was not made because of a message receipt from p_j. For simplicity we use the term differential vector clock only if the interpretation is clear from the context.

The protocol for a process p_i to maintain its local data structures is described as the following rules. Each rule consists of certain actions p_i must take right before it executes a specific event.

Rule1 (R1): Right before an event is executed at p_i, p_i sets $VC_i[i] \leftarrow VC_i[i] + 1$, $LU_i[i][0] \leftarrow i$, and $LU_i[i][1] \leftarrow VC_i[i]$.

Rule2 (R2): When p_i sends a message msg to a process p_j, p_i updates $VC_i[i]$, $LU_i[i][0]$, and $LU_i[i][1]$ according to rule **R1**, constructs the set $msg \cdot VC = \emptyset$ as follows:

$\forall (k, c_k) \in VC_i$
  if $(LS_i[j] < LU_i[k][1]) \wedge (LU_i[k][0] \neq j)$
    $\wedge (k \neq j)$ **then**
    $msg \cdot VC \leftarrow msg \cdot VC \cup \{(k, VC_i[k])\}$;

and attaches $msg \cdot VC$ to the message. Finally, p_i sets $LS_i[j] \leftarrow VC_i[i]$ before the message is sent. If $(j, s_j) \notin LS_i$, the operation $LS_i[j] \leftarrow VC_i[i]$ becomes $LS_i \leftarrow LS_i \cup \{(j, VC_i[i])\}$.

The set $msg \cdot VC$ is the differential vector clock of p_i relative to p_j and is attached to the message msg. The condition $LU_i[k][0] = j$ indicates that $VC_i[k]$ was modified due to a message receipt from p_j, while $LS_i[j] < LU_i[k]$ means that $VC_i[k]$ has been updated since last time p_i sent a message to p_j. In addition, we do not need to transfer $VC_i[j]$ to p_j because p_j has already known this. Therefore $msg \cdot VC$ is constructed to contain the elements of VC_i that have been updated since last time process p_i sent a message to p_j except (1) this modification was made because of a message receipt from p_j and (2) the element $VC_i[j]$. Note that this exception is an improvement over Sing-

hal and Kshemkalyani's implementation [12, 15].

Rule3 (R3): When p_i receives a message msg from process p_j, p_i extracts $msg \cdot VC$ from the message. Then p_i executes the actions that are described in Table 1. First, $VC_i[i]$ is incremented by one and $LU_i[i][0]$ and $LU_i[i][1]$ are updated. Then $VC_i[k]$, $LU_i[k][0]$, and $LU_i[i][1]$ are modified if $msg \cdot VC[k] > VC_i[k]$ holds. $LU_i[k][0]$ contains the ID of the message sender and $LU_i[k][1]$ is set to the updated value of $VC_i[i]$ if $VC_i[k]$ gets modified. Immediately

Table 1. Actions for p_i upon a message receipt from p_j

$VC_i[i] \leftarrow VC_i[i] + 1;$
$LU_i[i][0] \leftarrow i;$
$LU_i[i][1] \leftarrow VC_i[i];$

$\forall (k, c_k) \in msg \cdot VC$
  if $msg \cdot VC[k] > VC_i[k]$ **then**
    $VC_i[k] \leftarrow msg \cdot VC[k];$
    $LU_i[k][0] \leftarrow j;$
    $LU_i[k][1] \leftarrow VC_i[i];$

after taking those actions specified in the rules, process p_i timestamps the corresponding event with the value of VC_i, which can be used to keep track of causality relationships between the distributed events. If the logged events will be checked one by one in the order as they have been logged, then only the differences from the last logged timestamp need to be stored.

3.3. Process Creation and Termination

To join a system, an external process sends a joining request message to an existing process, the latter may accept this external process according to certain prescribed rules that will not be explained here. After a process is created or accepted, this new process will inherit the current value of the local data structures maintained at the process that creates or accepts the new one. When a process p_i terminates, the current value of its local data structures will be taken over by the process that created or accepted p_i or by one of its ancestors. This is done by running a process creation and termination protocol described in Table 2. We assume that an initial process p_i starts with $Leaving_i = false$.

As shown in Table 2, when a process p_i creates a new process or accepts an external process p_j, p_j becomes a child of p_i and the current value of VC_i is sent to p_j. Process p_j inherits the current value of VC_i, LU_i, and LS_i from p_i. Since p_j is created by p_i, p_i becomes the parent of p_j. TVC_j is set to an empty set because p_j is a new process and therefore no termination has been reported yet to p_j.

The procedure for a process to terminate is a little more complex. Before a process p_j terminates, it transfers the current value of VC_j, TVC_j, and $Child_j$ to p_j's parent through a $Transfer$ message and notifies p_j's children their new parent by sending them a $NewParent$ message. Note that TVC_j contains the vector clocks that were maintained at those processes who have reported termination to p_j. Upon knowing that p_j is leaving, p_j's parent, p_i, takes over the causality dependence transferred from p_j by recording this information in p_i's local data structure TCV_i if p_i is not leaving. Then p_i sends an acknowledgment back to p_j. Process p_j terminates when it receives an $AckTransfer$ message from its parent. The complexity of this procedure arises from the situation in which while p_j is leaving its parent may also be leaving. This situation is learned by p_j when it receives a $NewParent$ message from its current parent. In this case p_j will send a $Transfer$ message again to its new parent and notify its children their new parent. These actions repeat until p_j receives an acknowledgment for its $Transfer$ message from its parent and then terminates.

4. Correctness Arguments

In this section we will prove the properties of vector times that are useful in capturing causality relationships between distributed events.

Table 2. Actions for process creation and termination

Process Creation:

When p_i creates or accepts p_j, do the following:

$Child_i \leftarrow Child_i \cup \{j\}$;
send a message $Init(VC_i, LU_i, LS_i)$ to p_j;

When p_j receives a message $Init(VC_i, LU_i, LS_i)$ from p_i, do the following:

$Parent_j \leftarrow i$;
$Child_j \leftarrow \emptyset$;
$Leaving_j \leftarrow false$;
$VC_j \leftarrow VC_i \cup \{(j, 0)\}$;
$LU_j \leftarrow LU_i$;
$LU_j[j][0] \leftarrow j$;
$LU_j[j][1] \leftarrow 0$;
$LS_j \leftarrow LS_i \cup \{(j, 0)\}$;
$TVC_j \leftarrow \emptyset$;

Process Termination:

//We assume $Parent_j \neq j$, otherwise p_j cannot terminate.

When p_j terminates, do the following:

$Leaving_j \leftarrow true$;
send a message $Transfer(TVC_j \cup \{VC_j\}, Child_j)$ to process $Parent_j$;
$\forall k \in Child_j$, send a message $NewParent(Parent_j)$ to process k;

When p_i receives a $Transfer(TVC, Child)$ message from p_j, do the following:

if $(Leaving_i = false) \vee (Leaving_i = true \wedge Parent_i = j \wedge i < j)$ **then**
  $TVC_i \leftarrow TVC_i \cup TVC$; //$p_i$ is not allowed to terminate if
  $Child_i \leftarrow Child_i \cup Child$; //$p_i$ and p_j are the only two initial
  send an $AckTransfer()$ message to p_j; //processes and $i < j$ to ensure that
  if $Leaving_i = true$ **then** //at least one initial process exists.
    $Leaving_i = false$;
else ignore this message;

When p_i receives a $NewParent(Parent)$ message from p_j, do the following:

$Parent_i \leftarrow Parent$;
if $Parent_i = i \wedge i > j \wedge Leaving_i = true$ **then**
  clean up all local variables and terminate;
else if $Leaving_i = true$ **then**
  $\forall k \in Child_i$, send a message $NewParent(Parent_i)$ to process k;
  if $Parent_i = i$ **then**
    $Leaving_i \leftarrow false$;
  else
    send a message $Transfer(TVC_i \cup \{VC_i\}, Child_i)$ to process $Parent_i$;

When p_i receives an $AckTransfer()$ message from process $Parent_i$, do the following:

clean up all local variables and terminate;

Lemma 4.1 *When process p_i receives a computation message msg from p_j, the differential vector clock $msg \cdot VC$ contains all of the elements of VC_j whose value may be greater than that of the corresponding elements of VC_i at the moment when the message was sent.*

Proof This follows directly from the rules described in **R1**, **R2**, and **R3** for maintaining the data structures at each process, constructing a differential vector clock, and the requirement to attach the differential vector clock to the message from p_j to p_i.

When p_j sends a message to p_i, the condition $VC_j[k] > VC_i[k]$ could hold if and only if $VC_j[k]$ has been updated since last time a message was sent from p_j to p_i or since p_j was initiated if p_j has never sent a message to p_i. In addition, if this modification was made because of a message receipt from p_i, then the condition $VC_j[k] > VC_i[k]$ will not hold according to rule **R3** because in this case the condition $VC_j[k] \leq VC_i[k]$ must be satisfied. In other words, the elements of VC_j that may satisfy the condition $VC_j[k] > VC_i[k]$ include only those of them that have been modified since last time p_j sent a message to p_i and the modification was not done because of a message receipt from p_i. ∎

Theorem 4.2 *Let e and e' be two events, $e \rightarrow e' \iff VC(e) < VC(e')$.*

Proof This follows directly from Lemma 4.1 and and the proofs by Fidge [4] and Mattern [10]. ∎

Theorem 4.3 *When a process p_j is created or accepted by p_i, p_j inherits the current value of VC_i at the moment p_i creates or accepts p_j.*

Proof This follows directly from the process creation protocol described in Table 2 and the assumption of an FIFO channel. ∎

Theorem 4.4 *When a process p_j terminates, another process p_i in the system will take over the current value of VC_j at the moment when p_j terminates.*

Proof This follows directly from the process termination protocol described in Table 2 and the assumption of FIFO channels ∎

5. Efficiency Analysis

As proposed by Singhal and Kshemkalyani [15], we define the *efficiency* of the proposed technique as the average percentage reduction in the size of vector clock related information to be transferred with a message as compared to when sending the entire vector. The following terms are defined for this purpose:

A_p: The average number of entries in a differential clock that are transferred with a message using the proposed technique.

A_s: The average number of entries in a vector clock that are qualified for transmission with a message when the technique proposed by Singhal and Kshemkalyani [15] is used.

According to the rules to construct the differential clock for transmission in a message, the following inequality holds.

$$A_p \leq A_s. \tag{1}$$

The modification of an element of VC_i is made only because of (1) an internal event or a message sending event at p_i, (2) a message receipt from a process other than p_j, and (3) a message receipt from p_j. In Singhal and Kshemkalyani's proposal, all of the modifications are included in the differential vector clock, while the proposed technique includes only the modifications made in the first two cases. Equation (1) proves the improvements of our implementation over Singhal and Kshemkalyani's protocol [15].

In addition, the following inequality holds.

$$A_p \leq n \tag{2}$$

B_s: The number of bits to code the value of $VC_i[j]$.

B_p: The number of bits that are needed to code a process ID. Assuming that a process ID is represented with an integer number, then $B_p = log_2 n$.

When a vector clock is attached to a message, the

elements of this vector need to be identified even if the entire vector is transferred. The number of bits for each entry of the vector is $(B_p + B_s)$ and it is the same as when the entire vector is transferred in a dynamic system. Therefore the efficiency of the differential technique (E) is defined as follows:

$$\begin{aligned} E &= \left(1 - \frac{(B_p + B_s) \times A_p}{(B_p + B_s) \times n}\right) \times 100\% \\ &= \left(1 - \frac{A_p}{n}\right) \times 100\%. \end{aligned} \quad (3)$$

From equation (3) we know that the differential technique is always beneficial because of equation (2).

6. Conclusions

We have developed a differential technique to implement vector clocks in dynamic systems. The implementation is an extension of Singhal and Kshemkalyani's protocol [12, 15] and is theoretically more efficient than their protocol in reducing the required communication overhead. Correctness of the proposed technique has been proved.

When a process p_j is created or accepted by p_i, p_j inherits p_i's vector clock with the value when p_i creates or accepts p_j. When a process p_k terminates, some process in the system will take over p_k's vector clock with the value when p_k terminates. These actions of inheritance and takeover are guaranteed even when several processes terminate concurrently.

References

[1] R. Baldoni and M. Raynal. Fundamentals of distributed computing: A practical tour of vector clock systems. *IEEE Distributed Systems Online*, 3(2), 2002.

[2] B. Charron-Bost. Concerning the size of logical clocks in distributed systems. *Information Processing Letters*, 39(1):11–16, 12 July 1991.

[3] C. J. Fidge. Timestamps in message-passing systems that preserve partial ordering. In *Proceedings of the 11th Australian Computer Science Conference*, pages 56–66, Feb. 1988.

[4] C. J. Fidge. Logical time in distributed computing systems. *IEEE Computer*, 24(8):28–33, 1991.

[5] C. J. Fidge. Fundamentals of distributed system observation. *IEEE Software*, 13(6):77–83, 1996.

[6] C. J. Fidge. A limitation of vector timestamps for reconstructing distributed computations. *Information Processing Letters*, 66(2):87–91, 1998.

[7] V. K. Garg and C. Skawratananond. Timestamping messages in synchronous computations. In *Proceedings of the 22nd International Conference on Distributed Computing Systems*, pages 552 – 559. IEEE Computer Society Press, 2002.

[8] J. M. Hélary, M. Raynal, G. Melideo, and R. Baldoni. Efficient causality-tracking timestamping. *IEEE Transactions on Knowledge and Data Engineering*, 15(5):1239–1250, 2003.

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

[10] F. Mattern. Virtual time and global states of distributed systems. In M. Cosnard and et al., editors, *Parallel and Distributed Algorithms: Proceedings of International Workshop on Parallel and Distributed Algorithms*, pages 215–226. Elsevier Science Publishers B. V., North-Holland, 1989.

[11] M. Raynal and M. Singhal. Logical time: A way to capture causality in distributed systems. Technical Report RR-2472.

[12] M. Raynal and M. Singhal. Logical time: capturing causality in distributed systems. *IEEE Computer*, 29(2):49–56, 1996.

[13] G. G. Richard III. Efficient vector time with dynamic process creation and termination. *Journal of Parallel and Distributed Computing*, 55(1):109–120, 1998.

[14] F. Ruget. Cheaper matrix clocks. In G. Tel and P. M. B. Vitányi, editors, *Proceedings of the 8th International Workshop on Distributed Algorithms (WDAG94)*, volume 857, pages 355–369, Terschelling, The Netherlands, 29 –1 1994. Springer-Verlag.

[15] M. Singhal and A. Kshemkalyani. An efficient implementation of vector clocks. *Information Processing Letters*, 43(1):47–52, August 1992.

[16] F. J. Torres-Rojas and M. Ahamad. Plausible clocks: Constant size logical clocks for distributed systems. In *Proceedings of the Workshop on Distributed Algorithms (WDAG)*, pages 71–88, 1996.

View publication stats