On Reducing the Complexity of Matrix Clocks

Lúcia M. A. Drummond

Universidade Federal Fluminense
Instituto de Computação
Rua Passo da Pátria, 156
24210-240 Niterói - RJ, Brazil
lucia@dcc.ic.uff.br

Valmir C. Barbosa

Universidade Federal do Rio de Janeiro
Programa de Engenharia de Sistemas e Computação, COPPE
Caixa Postal 68511
21945-970 Rio de Janeiro - RJ, Brazil
valmir@cos.ufrj.br

Abstract

Matrix clocks are a generalization of the notion of vector clocks that allows the local representation of causal precedence to reach into an asynchronous distributed computation's past with depth x, where $x \geq 1$ is an integer. Maintaining matrix clocks correctly in a system of n nodes requires that every message be accompanied by $O(n^x)$ numbers, which reflects an exponential dependency of the complexity of matrix clocks upon the desired depth x. We introduce a novel type of matrix clock, one that requires only nx numbers to be attached to each message while maintaining what for many applications may be the most significant portion of the information that the original matrix clock carries. In order to illustrate the new clock's applicability, we demonstrate its use in the monitoring of certain resource-sharing computations.

Keywords: Vector clocks, matrix clocks.

1. Introduction and background

We consider an undirected graph G on n nodes. Each node in G stands for a computational process and undirected edges in G represent the possibilities for bidirectional point-to-point communication between pairs of processes. A distributed computation carried out by the distributed system represented by G can be viewed as a set of events occurring at the various nodes. An event is the sending of a message by a node to another node to which it is directly connected in G (a neighbor in G), or the reception of a message from such a neighbor, or yet the occurrence at a node of any internal state transition of relevance ("state" and "relevance" here are highly dependent upon the particular computation at hand, and are left unspecified).

The standard framework for analyzing such a system is the partial order, often denoted by $\prec$, that formalizes the usual "happened-before" notion of distributed computing [1, 10]. This partial order is the transitive closure of the more elementary relation to which the ordered pair (v, v') of events belongs if either v and v' are consecutive events at a same node in G or v and v' are, respectively, the sending and receiving of a message between neighbors in G.

At node i, we let local time be assessed by the number t_i of events that already occurred. We interchangeably adopt the notations $t_i = time_i(v)$ and $v = event_i(t_i)$ to indicate that v is the t_ith event occurring at node i, provided $t_i \geq 1$. Another important partial order on the set of events is the relation that gives the predecessor at node $j \neq i$ of an event v' occurring at node i. We say that an event v is such a predecessor, denoted by $v = pred_j(v')$, if v is the event occurring at node j such that $v \prec v'$ for which $time_j(v)$ is greatest. If no such v exists, then $pred_j(v')$ is undefined and $time_j\big(pred_j(v')\big)$ is assumed to be zero.

The relation $pred_j$ allows the definition of vector clocks [8, 9, 12], as follows. The vector clock of node i at time t_i (that is, following the occurrence of t_i events at node i), denoted by $V^i(t_i)$, is a vector whose jth component, for $1 \leq j \leq n$, is either equal to 0 (if $t_i = 0$) or given by

$$V^i_j(t_i) = \begin{cases} t_i, & \text{if } j = i; \\ time_j\Big(pred_j\big(event_i(t_i)\big)\Big), & \text{if } j \neq i \end{cases} \tag{1}$$

(if $t_i \geq 1$). In other words, $V^i_j(t_i)$ is either the current time at node i, if $j = i$, or is the time at node j that results from the occurrence at that node of the predecessor of the t_ith event of node i, otherwise. If no such event exists (i.e., $t_i = 0$), then $V^i_j(t_i) = 0$.

Vector clocks evolve following two simple rules:

- Upon sending a message to node k, node i attaches $V^i(t_i)$ to the message, where t_i is assumed to already account for the message that is being sent.
- Upon receiving a message from node k with attached vector clock V^k, node i sets $V^i_i(t_i)$ to t_i (which is assumed to already reflect the reception of the message) and $V^i_j(t_i)$ to $\max\{V^i_j(t_i - 1), V^k_j\}$, for $j \neq i$.

It is a simple matter to verify that these rules do indeed maintain vector clocks consistently with their definition [8, 9, 12]. Under these rules or variations thereof, vector clocks have proven useful in a variety of distributed algorithms to detect some types of global predicates [7, 9].

One natural generalization of the notion of vector clocks is the notion of matrix clocks [14, 16]. For an integer $x \geq 1$, the x-dimensional matrix clock of node i at time t_i, denoted by $M^i(t_i)$, has $O(n^x)$ components. For $1 \leq j_1, \ldots, j_x \leq n$, component $M^i_{j_1,\ldots,j_x}(t_i)$ of $M^i(t_i)$ is only defined for $i = j_1 = \cdots = j_x$ and for $i \neq j_1 \neq \cdots \neq j_x$. As in the definition of $V^i(t_i)$, $M^i_{j_1,\ldots,j_x}(t_i) = 0$ if $t_i = 0$. For $t_i \geq 1$, on the other hand, we have

$$M^i_{j_1,\ldots,j_x}(t_i) = \begin{cases} t_i, & \text{if } i = j_1 = \cdots = j_x; \\ time_{j_x}\Big(pred_{j_x} \ldots pred_{j_1}\big(event_i(t_i)\big)\Big), & \text{if } i \neq j_1 \neq \cdots \neq j_x, \end{cases} \tag{2}$$

which, for $i \neq j_1 \neq \cdots \neq j_x$, first takes the predecessor at node j_1 of the t_ith event occurring at node i, then the predecessor at node j_2 of that predecessor, and so on through node j_x, whose local time after the occurrence of the last predecessor in the chain is assigned to $M^i_{j_1,\ldots,j_x}(t_i)$. It is straightforward to see that, for $x = 1$, this definition is equivalent to the definition of a vector clock in (1). Similarly, the maintenance of matrix clocks follows rules entirely analogous to those used to maintain vector clocks [9].

While the jth component of the vector clock following the occurrence of event v' at node $i \neq j$ gives the time resulting at node j from the occurrence of $pred_j(v')$, the analogous interpretation that exists for matrix clocks requires the introduction of additional notation. Specifically, the definition of a set of events encompassing the possible x-fold compositions of the relation $pred_j$ with itself, denoted by $Pred^{(x)}_j$, is necessary. If an event v occurs at node j, then we say that $v \in Pred^{(x)}_j(v')$ for an event v' occurring at node i if one of the following holds:

- $x = 1$, $j \neq i$, and $v = pred_j(v')$.
- $x > 1$ and there exists $k \neq i$ such that an event $\bar{v}$ occurs at node k for which $\bar{v} = pred_k(v')$ and $v \in Pred^{(x-1)}_j(\bar{v})$.

Note that this definition requires $j \neq k$, in addition to $k \neq i$, for $v \in Pred^{(x)}_j(v')$ to hold when $x = 2$.

For $x > 1$ and $j_1 \neq \cdots \neq j_x = j$, this definition allows the following interpretation of entry $j_1, \ldots, j_x$ of the matrix clock that follows the occurrence of event v' at node i, that is, of $M^i_{j_1,\ldots,j_{x-1},j}\big(time_i(v')\big)$. It gives the time resulting at node j from the occurrence of an event that is in the set $Pred^{(x)}_j(v')$. Of course, the number of possibilities for such an event is $O(n^{x-1})$ in the worst case, owing to the several possible combinations of $j_1, \ldots, j_{x-1}$.

Interestingly, applications that do require the full capabilities of matrix clocks may be still unknown. In fact, it seems a simple matter to argue that a slightly more sophisticated use of vector clocks or simply the use of two-dimensional matrix clocks suffices to tackle some of the problems that have been offered as possible applications of multidimensional matrix clocks [9, 13]. What we do in this paper is to demonstrate how the distributed monitoring of certain resource-sharing computations can benefit from the use of matrix clocks and that, at least for such computations, it is possible to employ much less complex matrix clocks (that is, matrix clocks with many fewer components), which nonetheless retain the ability to reach into the computation's past with arbitrary depth.

The key to this reduction in complexity is the use of one single event in place of the set $Pred^{(x)}_j(v')$. As we argue in Section 3, following a brief discussion of the resource-sharing context in Section 2, this simplification leads to matrix clocks of size nx, therefore considerably less complex than the $O(n^x)$-size original matrix clocks. Concluding remarks follow in Section 4, after a discussion of how the technique of Section 3 can successfully solve the problem posed in Section 2.

2. Monitoring resource-sharing computations

The resource-sharing computation we consider is one of the classical solutions to the paradigmatic Dining Philosophers Problem (DPP) [6] in generalized form. In this case, G's edge set is constructed from a given set of resources and from subsets of that set, one for each node, indicating which resources can be ever needed by that node. This construction places an edge between nodes i and j if the sets of resources ever to be needed by i and j have a nonempty intersection. Notice that this construction is consonant with the interpretation of edges as bidirectional communication channels, because what it does is to deploy edges between every pair of nodes that may ever compete for a same resource and must therefore be able to communicate with each other to resolve conflicts.

In DPP, the computation carried out by a node makes it cycle endlessly through three states, which are identified with the conditions of being idle, being in the process of acquiring exclusive access to the resources it needs, and using those resources for a finite period of time. While in the idle state, the node starts acquiring exclusive access to resources when the need arises to compute on shared resources. It is a requirement of DPP that the node must acquire exclusive access to all the resources it shares with all its neighbors, so it suffices for the node to acquire a token object it shares with each of its neighbors (the "fork," as it is called), each object representing all the resources it shares with that particular neighbor. When in possession of all forks, the node may start using the shared resources [1, 5].

The process of collecting forks from neighbors follows a protocol based on the sending of request messages by the node that needs the forks and the sending of the forks themselves by the nodes that have them. More than one protocol exists, each implementing a different rule to ensure the absence of deadlocks and lockouts during the computation. The solution we consider in this section is based on relative priorities assigned to nodes. Another prominent solution is also based on the assignment of relative priorities, but to the resources instead of to the nodes [11, 15].

The priority scheme of interest to us is based on the graph-theoretic concept of an acyclic orientation of G, which is an assignment of directions to the edges of G in such a way that directed cycles are not formed. Such an acyclic orientation is then a partial order of the nodes of G, and is as such suitable to indicate, given a pair of neighbors, which of the two has priority over the other. Most of the details of this priority scheme are not relevant to our present discussion, but before stating its essentials we do mention that the lockout-freedom requirement leads the acyclic orientation to be changed dynamically (so that relative priorities are never fixed), which in turn leads to a rich dynamics in the set of all the acyclic orientations of G and to important concurrency-related issues [2–4].

Once a priority scheme is available over the set of nodes, what matters to us is how it is used in the fork-collecting protocol. When a request for fork arrives at node j from node i, j sends i the fork they share if j is either idle or is also collecting forks but does not have priority over i. If j is also collecting forks and has priority over i, or if j is using shared resources, then the sending of the fork to i is postponed to until j has finished using the shared resources. Note that two types of wait may happen here. If j is using shared resources when the request arrives, then the wait is independent of n. If j is also collecting forks, then the wait for j to start using the shared resources and ultimately to send i the fork is in the worst case $n-1$ [1, 3, 5]. The reason for this is simple: j is waiting for a fork from another node, which may in turn be waiting for a fork from yet another node, and so on. Because the priority scheme is built on acyclic orientations of G, such a chain of waits does necessarily end and is $n-1$ nodes long in the worst case.

Whether such long waits occur or not is of course dependent upon the details of each execution of the resource-sharing computation. But if they do occur, one possibility for reducing the average wait is to increase the availability of certain critical resources so that G becomes less dense [1, 3, 4]. Perhaps another possibility would be to fine-tune some of the characteristics of each node's participation in the overall computation, such as the duration of its idle period, for example. In any event, the ability to locally detect long waits (a global property, since it relates to the notion of time in fully asynchronous distributed systems [1]) is crucial.

To see how this relates to the formalism of Section 1, suppose our resource-sharing computation consists of the exchange of fork-bearing messages only (that is, request messages and all other messages involved in the computation, such as those used to update acyclic orientations, are ignored). For distinct nodes i and j, and for an event v' occurring at node i at the reception of a fork, the set $Pred_j^{(x)}(v')$ for $x \geq 1$ is either empty or only contains events that correspond to the sending of forks by node j. Now suppose we systematically investigate the sets $Pred_j^{(x)}(v')$ for every appropriate j by letting x increase from 1 through $n-1$. Suppose also that, for each j, we record the first x that is found such that $Pred_j^{(x)}(v')$ contains the sending of a fork as response to the reception of a request message without any wait for fork collection on the part of j. The greatest such value to be recorded, say x^*, has special significance: it means that the eventual reception of a fork by node i through v' is the result of an x^*-long chain of waits.

The matrix clock that we introduce in Section 3 is capable of conveying this information to node i succinctly, so long as there exists enough redundancy in the sets $Pred_j^{(x)}(v')$ that attaching only nX integers to forks suffices, where X is a threshold in the interval $[1, n-2]$ beyond which wait chains are considered too long. It so happens that such redundancy clearly exists: the only events in $Pred_j^{(x)}(v')$ that matter are those corresponding to the sending of forks without any wait for fork collection. Detecting any one of them suffices, so we may as well settle for the latest, that is, one single event from the whole set. We return to this in Section 4.

3. A simpler matrix clock

For $x \geq 1$, the new matrix clock we introduce is an $x \times n$ matrix. For node i at time t_i (i.e., following the occurrence of t_i events at node i), it is denoted by $C^i(t_i)$. For $1 \leq y \leq x$ and $1 \leq j \leq n$, component $C^i_{y,j}(t_i)$ of $C^i(t_i)$ is defined as follows. If $t_i = 0$, then $C^i_{y,j}(t_i) = 0$, as for vector clocks and the matrix clocks of Section 1. If $t_i \geq 1$, then we have

$$C^i_{y,j}(t_i) = \begin{cases} t_i, & \text{if } y = 1 \text{ and } j = i; \\ \max_{i \neq j_1 \neq \cdots \neq j_y = j} \Big\{ time_{j_y} \Big(pred_{j_y} \ldots pred_{j_1} \left(event_i(t_i) \right) \Big) \Big\}, & \text{if } y > 1 \text{ or } j \neq i. \end{cases} \quad (3)$$

Note, first of all, that for $y = 1$ this definition is equivalent to the definition of a vector clock in (1). Thus, the first row of $C^i(t_i)$ is the vector clock $V^i(t_i)$. For $y > 1$, the definition in (3) implies, according to the interpretation of matrix clocks that follows our definition in (2), that

$$C^i_{y,j}(t_i) = \max\{\mathit{time}_j(v) \mid v \in \mathit{Pred}^{(y)}_j(v')\}, \tag{4}$$

where v' is the t_ith event occurring at node i. What this means is that, of all the $O(n^{y-1})$ events that may exist in $\mathit{Pred}^{(y)}_j(v')$, only one (the one to have occurred latest at node j) makes it to the matrix clock $C^i(t_i)$. Note also that (3) implies the equality in (4) for $y = 1$ as well, so long as $j \neq i$. In this case, $\mathit{Pred}^{(y)}_j(v')$ is the singleton $\{\mathit{pred}_j(v')\}$.

Now, we know from the definition of $\mathit{Pred}^{(y)}_j$ that, for $y > 1$, $v \in \mathit{Pred}^{(y)}_j(v')$ if and only if there exists $k \neq i$ such that an event $\bar{v}$ occurs at node k for which $\bar{v} = \mathit{pred}_k(v')$ and $v \in \mathit{Pred}^{(y-1)}_j(\bar{v})$. Thus, if we assume that $t_k = \mathit{time}_k(\bar{v})$ and consider the matrix clocks resulting from the occurrence of $\bar{v}$ and of v', that is, $C^k(t_k)$ and $C^i(t_i)$, respectively, then it is a consequence of (4) that

$$C^i_{y,j}(t_i) \geq C^k_{y-1,j}(t_k),$$

where the concomitant occurrence of $j = k$ and $y = 2$ is automatically precluded. Letting $K^i(t_i, y, j)$ be the set comprising every appropriate k, we then obtain

$$C^i_{y,j}(t_i) = \begin{cases} \max_{k \in K^i(t_i,y,j)} C^k_{y-1,j}(t_k), & \text{if } K^i(t_i,y,j) \neq \emptyset; \\ 0, & \text{if } K^i(t_i,y,j) = \emptyset \end{cases} \tag{5}$$

for $y > 1$.

By (5), we are then left with the following two rules for the evolution of our new matrix clocks:

- Upon sending a message to node k, node i attaches $C^i(t_i)$ to the message, where t_i is assumed to already account for the message that is being sent.
- Upon receiving a message from node k with attached matrix clock C^k, node i sets $C^i_{1,i}(t_i)$ to t_i (which is assumed to already reflect the reception of the message) and $C^i_{y,j}(t_i)$ to $\max\{C^i_{y,j}(t_i - 1), C^k_{y-1,j}\}$, for $1 < y \leq x$ or $j \neq i$, provided $y > 2$ or $j \neq k$. For $y = 2$ and $j = k$, $C^i_{y,j}(t_i)$ retains the value of $C^i_{y,j}(t_i - 1)$.

According to these rules, every message carries an attachment of nx integers.

4. Discussion and concluding remarks

We are then in position to return to the problem posed in Section 2, namely the problem of monitoring executions of the solution to DPP that employs a partial order on G's set of nodes to establish priorities. As we discussed in that section, the goal is to allow nodes to detect locally, upon receiving a fork, whether the delivery of that fork is the result of a chain of fork deliveries that started too far back in the past. In the affirmative case, the wait since the fork was requested will have been too long, in terms of the usual notions of time complexity in asynchronous distributed computations.

More specifically, if v' is the event corresponding to the reception of a fork at node i, then the goal is to be able to detect the existence of an event in $Pred_j^{(y)}(v')$ that corresponds to the sending of a fork by node j either immediately upon the reception of the request for that fork or, if node j was using shared resources when the request arrived, immediately upon finishing. Here $1 \leq y \leq X$ and j is any node, provided $y = 1$ and $j = i$ do not occur in conjunction. The value of X is such that $1 \leq X \leq n - 2$, and is chosen as a threshold to reflect the maximum allowable chain of waits. As a consequence, the sets $Pred_j^{(y)}(v')$ must include fork-related events only.

The new matrix clocks introduced in Section 3 can be used for this detection with only minimal adaptation. The key ingredients are:

- The only messages sent by node i to be tagged with the matrix clock C^i are forks. If the sending of a fork by node i does not depend on further fork collection by i, then C^i is reset to zero before it is attached to the fork. Matrix clocks are $X \times n$, and all further handling of them follows the rules given in Section 3.

- Upon receiving a fork with attached matrix clock, and having updated C^i accordingly, node i looks for components of C^i that contain the value zero. If none exists, then a wait chain that is considered too long has been found.

This strategy reflects the general method of Section 3 when applied to events that relate to the flow of forks only. Whenever the request for a fork is received and the fork can be sent without the need for any forks to be received by the node in question, zeroes get inserted into the matrix clock and are sent along with the fork. The sending of this fork may unleash the sending of forks by other nodes in a chain of events, and along the way those zeroes may get replaced by greater integers, reflecting the increasing length of the wait chain rooted at the original node. The reception of a fork whose matrix clock has no zeroes is then an indication that such reception is part of chains of length at least X.

In summary, we have in this paper introduced a novel notion of matrix clocks. Similarly to the matrix clocks originally introduced in [14, 16] and whose definition appears in (2), our matrix clock has the potential of reflecting causal dependencies in the flow of messages that stretch as far as depth x into the past. Unlike those original matrix clocks, however, ours increases with x as nx, while in the original case the growth is according to an $O(n^x)$ function, that is, exponentially.

We have illustrated the applicability of the new matrix clocks with an example from the area of resource-sharing problems. What we have demonstrated is a means of collecting information locally during the resource-sharing computation so that exceedingly long global waits can be detected, possibly indicating the need for overall system re-structuring.

Acknowledgments

The authors acknowledge partial support from CNPq, CAPES, the PRONEX initiative of Brazil's MCT under contract 41.96.0857.00, and a FAPERJ BBP grant.

References

1. V. C. Barbosa, *An Introduction to Distributed Algorithms*, The MIT Press, Cambridge, MA, 1996.

2. V. C. Barbosa, *An Atlas of Edge-Reversal Dynamics*, Chapman & Hall/CRC, London, UK, 2000.

3. V. C. Barbosa, "The combinatorics of resource sharing," in R. Corrêa *et alii* (Eds.), *Models for Parallel and Distributed Computation: Theory, Algorithmic Techniques and Applications*, Kluwer Academic Publishers, Dordrecht, The Netherlands, to appear.

4. V. C. Barbosa and E. Gafni, "Concurrency in heavily loaded neighborhood-constrained systems," *ACM Trans. on Programming Languages and Systems* **11** (1989), 562–584.

5. K. M. Chandy and J. Misra, "The drinking philosophers problem," *ACM Trans. on Programming Languages and Systems* **6** (1984), 632–646.

6. E. W. Dijkstra, "Hierarchical ordering of sequential processes," *Acta Informatica* **1** (1971), 115–138.

7. L. M. A. Drummond and V. C. Barbosa, "Distributed breakpoint detection in messge-passing programs," *J. of Parallel and Distributed Computing* **39** (1996), 153–167.

8. L. J. Fidge, "Timestamp in message passing systems that preserves partial ordering," *Proc. of the 11th Australian Computing Conference*, 56–66, 1988.

9. V. K. Garg, *Principles of Distributed Systems*, Kluwer Academic Publishers, Boston, MA, 1996.

10. L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Comm. of the ACM* **21** (1978), 558–565.

11. N. A. Lynch, "Upper bounds for static resource allocation in a distributed system," *J. of Computer and System Sciences* **23** (1981), 254–278.

12. F. Mattern, "Virtual time and global states in distributed systems," in M. Cosnard *et alii* (Eds.), *Parallel and Distributed Algorithms: Proc. of the Int. Workshop on Parallel and Distributed Algorithms*, 215–226, North-Holland, Amsterdam, The Netherlands, 1989.

13. M. Raynal, "Illustrating the use of vector clocks in property detection: an example and a counter-example," in P. Amestoy *et alii* (Eds.), *Euro-Par'99—Parallel Processing*, 806–814, Lecture Notes in Computer Science 1685, Springer-Verlag, Berlin, Germany, 1999.

14. S. K. Sarin and L. Lynch, "Discarding obsolete information in a replicated database system," *IEEE Trans. on Software Engineering* **SE-13** (1987), 39–46.

15. J. L. Welch and N. A. Lynch, "A modular drinking philosophers algorithm," *Distributed Computing* **6** (1993), 233–244.

16. G. T. J. Wuu and A. J. Bernstein, “Efficient solutions to the replicated log and dictionary problems,” *Proc. of the 3rd Annual ACM Symposium on Principles of Distributed Computing*, 233–242, 1984.